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

Michael Gattavecchia (n/m 0000362269)
michael.gatta@gmail.com

ALMA MATER STUDIORUM
II Facoltà di Ingegneria - Sede di Cesena
Linguaggi e Modelli Computazionali L-S

18 novembre 2010

Indice

Versioni a confronto

Prima fase di analisi, versioni a confronto:

- TuProlog 2.1*
- TuProlog 2.1 fast-match (indexing)
- TuProlog 2.3alpha*

Valutazione dei tempi medi per l'esecuzione di alcuni benchmark ben noti, e per altri realizzati ad-hoc.

*: versioni modificate come suggerito da Michele Damian (si veda la bibliografia)

Risultati dei benchmark

Benchmark	iterations	2.1 [ms]	2.3(alpha) [ms]	2.1 (f-m) [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: Tempi di esecuzione delle tre versioni a confronto

Asserting

Valutazione delle prestazioni in caso di pesante ed esclusivo ricorso ad asserzione.

Teoria

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

Goal

```
?-asserter2(n). % n intero
```

Asserting (cont.)

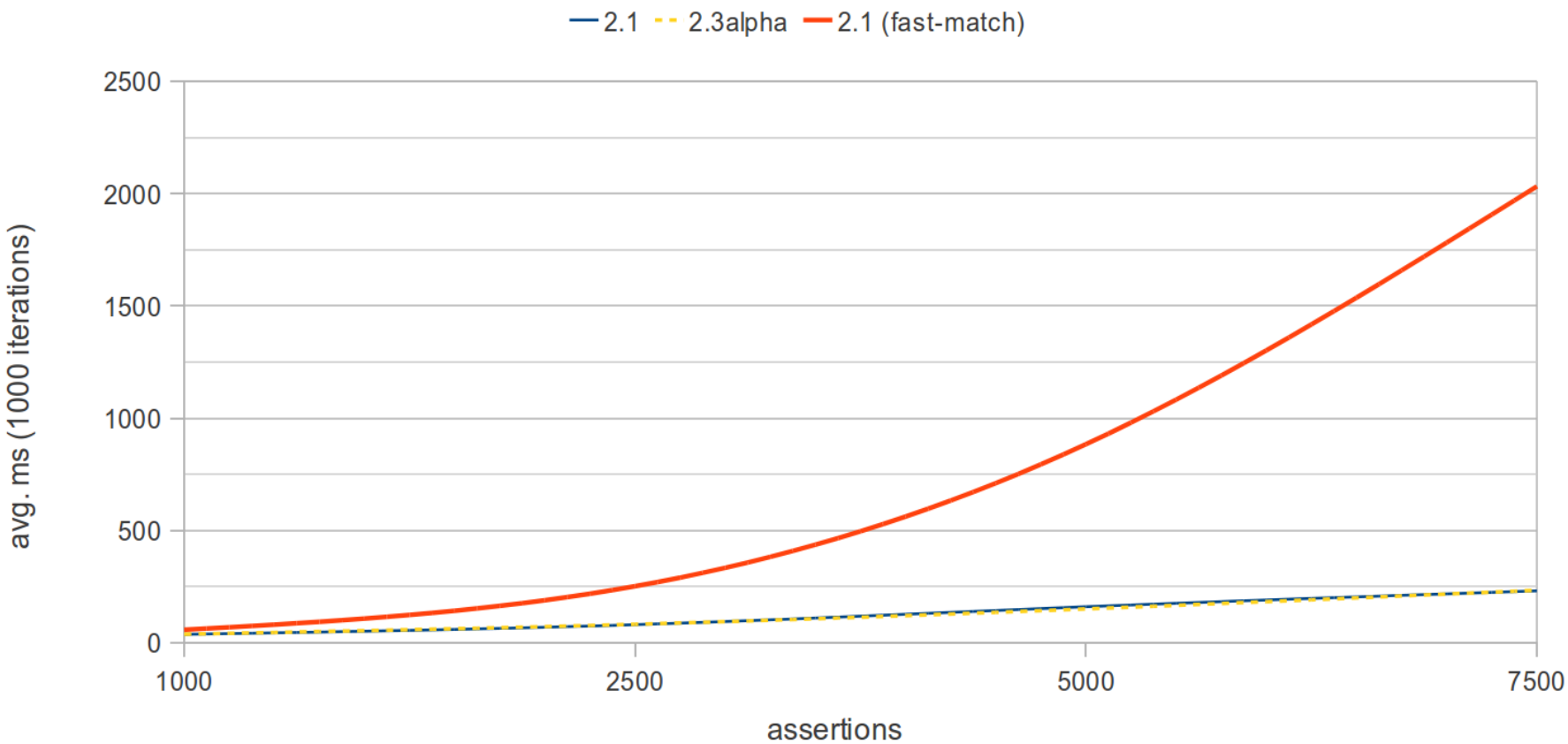

Figura: Tempi medi di esecuzione (su 1000 iterazioni) in funzione del numero di asserzioni

Asserting (cont.)

Dal profiling è emerso che la maggior parte dell'overhead computazionale era imputabile ai metodi:

- `RBTree.verifyProperty5Helper(Node, int, int)`
- `RBTree.verifyProperty4(Node)`
- `RBTree.noteColor(Node)`
- `RBTree.verifyProperty1(Node)`

Analizzando il codice dei Red-Black Tree introdotti da Contessi si è notata la presenza di un meccanismo di checking continuo del bilanciamento degli alberi.

È possibile disabilitarlo attraverso il setting a *false* del flag `VERIFY_RBTREE`

Asserting (cont.)

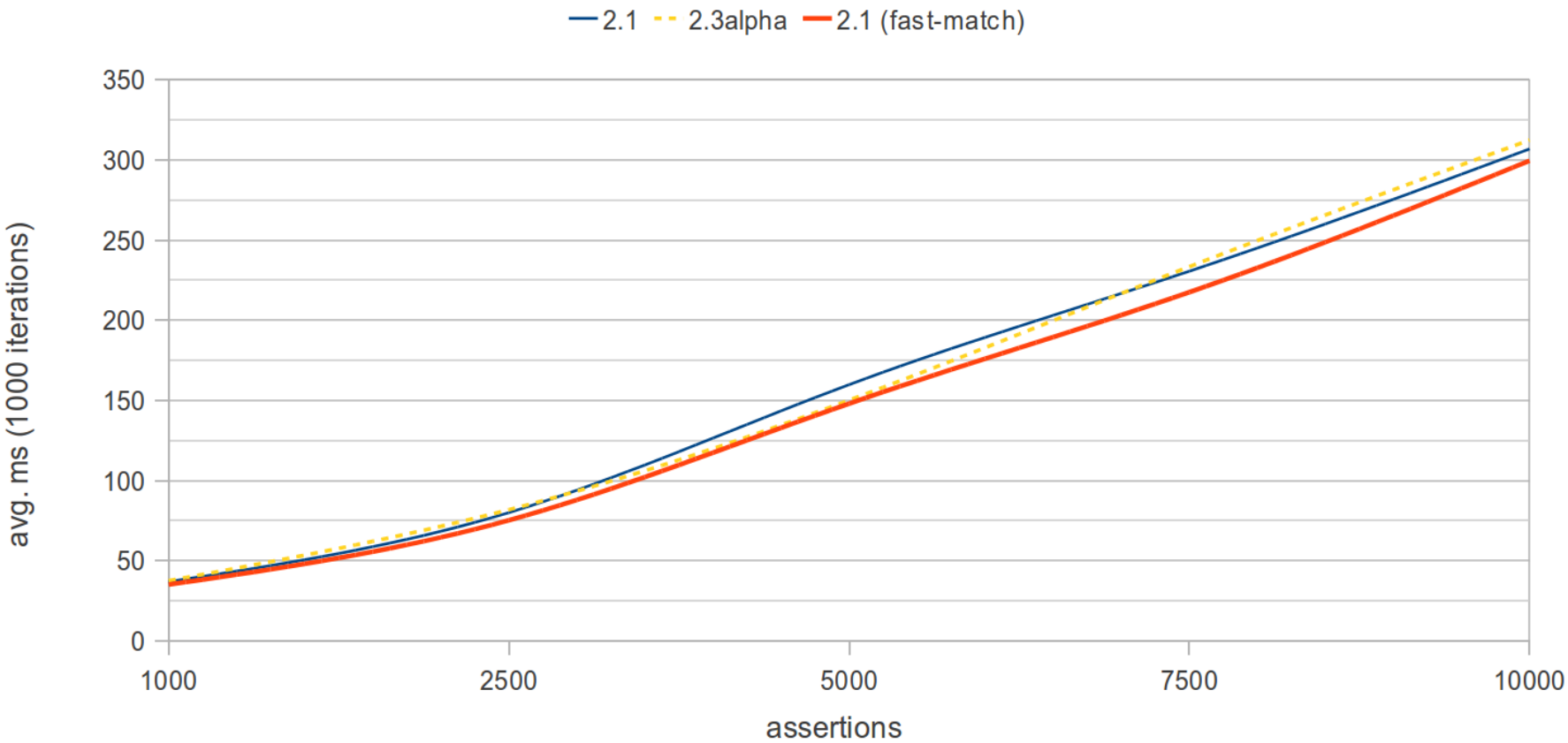

Figura: Tempi medi di esecuzione (su 1000 iterazioni) in funzione del numero di asserzioni (`VERIFY_RBTREE`=false)

Classe `Struct`

La risoluzione di un goal determina un grande numero di chiamate al metodo `getTerm(int)` della classe `Struct`

Term.getTerm(int)

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

La variabile `arg` è un array di `Term`, ma la classe `Term` è astratta, ed il metodo è definito all'interno delle sue varie specializzazioni.

Classe `Struct` (cont.)

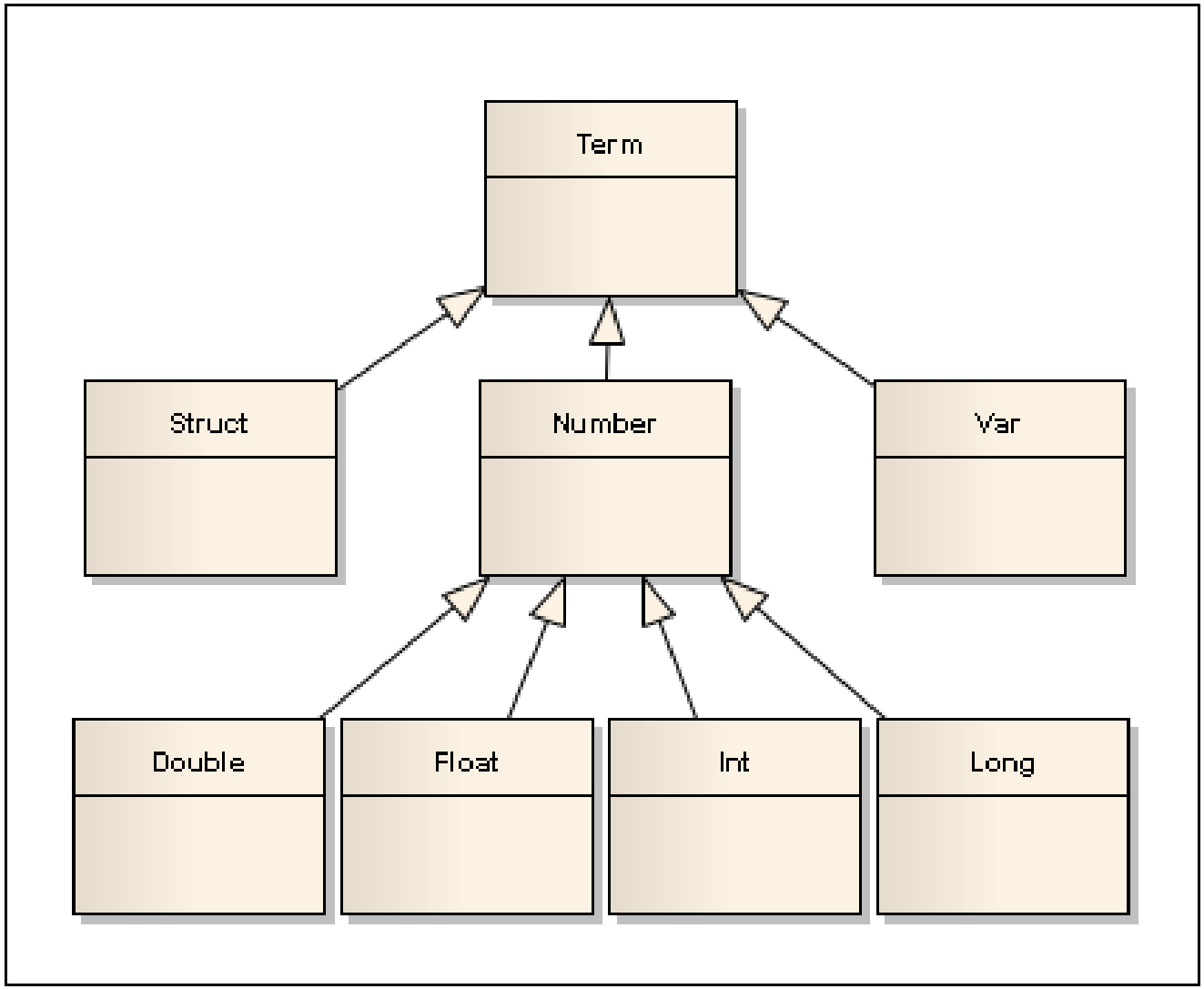

Figura: Class diagram relativo alla gerarchia al di sotto della classe `Term`

Classe `Struct` (cont.)

All'interno delle classi `Struct` e `Number` il metodo è definito in modo estremamente semplice:

metodo `getTerm()` nelle classi `Number` e `Struct`

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

mentre all'interno della classe `Var` il metodo presenta una struttura più articolata.

Classe `Struct` (cont.)

Si è quindi imposto un controllo preventivo sulla specifica instanza di `Term` sulla quale viene invocato il metodo `getTerm()`

metodo `Struct.getTerm(int)` modificato

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

in questo modo, nel caso in cui l'elemento del vettore in questione sia (per esclusione) instanza delle classi `Number` o `Struct` si evitano ulteriori chiamate inutili.

Classe `Struct` (cont.)

Il notevole numero di chiamate al metodo `Struct.getTerm(int)` ha determinato buoni improvement prestazionali rispetto al tempo totale di esecuzione.

Benchmark	iterations	avg (old) [ms]	avg (new) [ms]	impr.
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: Tempi di esecuzione delle due implementazioni a confronto

Complessivamente si è potuto constatare un miglioramento prestazionale medio (diminuzione dei tempi di esecuzione) maggiore del 3%.

TuProlog 2.3alpha

Le modifiche analizzate sono state interamente integrate sulla versione 2.3alpha di TuProlog (ultima versione disponibile).

Il codice di Paolo Contessi è stato leggermente modificato in un punto a causa dell'errato utilizzo di un iteratore.

La versione 2.3 alpha, dopo l'applicazione delle modifiche, supera con successo l'intera suite di test presente nelle precedenti versioni di TuProlog (`TuPrologTestSuite`).

TuProlog 2.3alpha - benchmarks

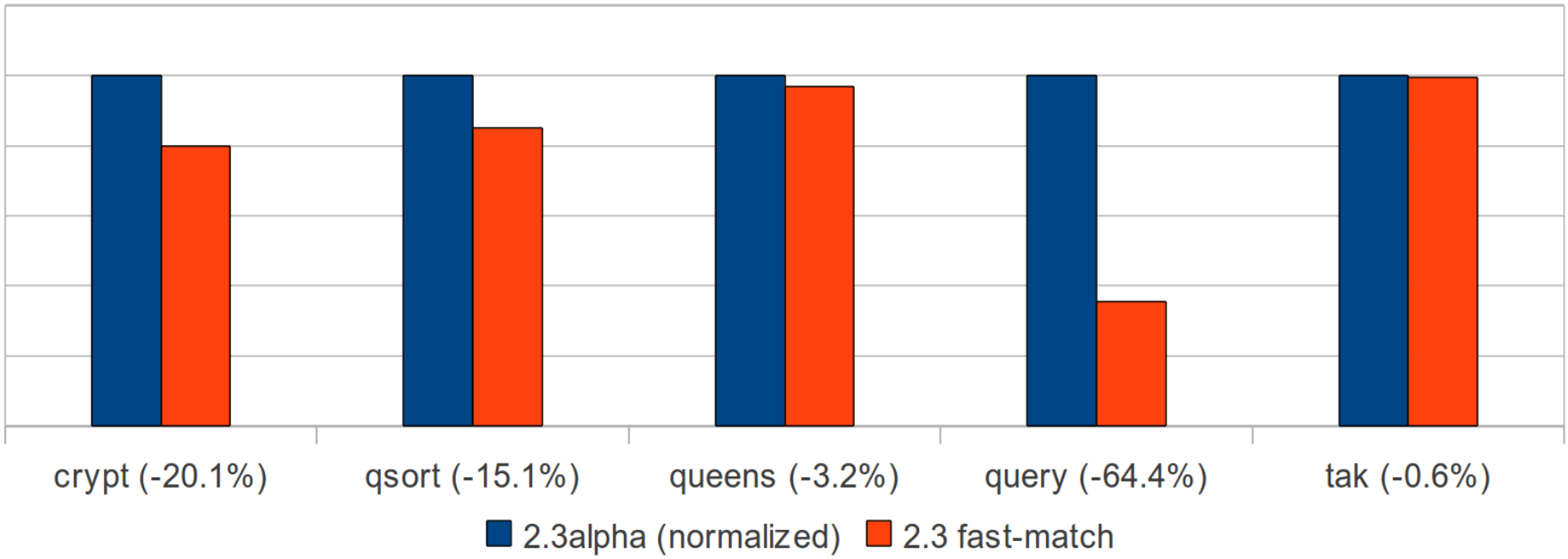

Figura: Benchmark tuProlog 2.3 dopo l'applicazione di fast-match e modiche proposte.

TuProlog 2.3alpha - benchmarks (cont.)

Benchmark	iterations	2.3alpha [ms]	2.3 (f-m) [ms]	impr.
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: Tempi di esecuzione delle due implementazioni (2.3.0alpha)

Le modifiche apportate hanno permesso di ottenere un improvement complessivo medio $\approx 20.5\%$

Tail Recursion Optimization

- Il motore alla base della risoluzione in TuProlog è un FSA.
- Le strutture dati utilizzate per la risoluzione sono alberi, che durante la risoluzione sono sostanzialmente riconducibili a stack.
- La TRO non è mai stata efficacemente implementata all'interno di TuProlog.

La TRO deve concretizzarsi evitando di dover "risalire" la catena risolutiva e restituendo al chiamante il risultato non appena esso diventa disponibile, ovviamente a patto che la teoria sia opportunamente predisposta.

TuProlog FSA

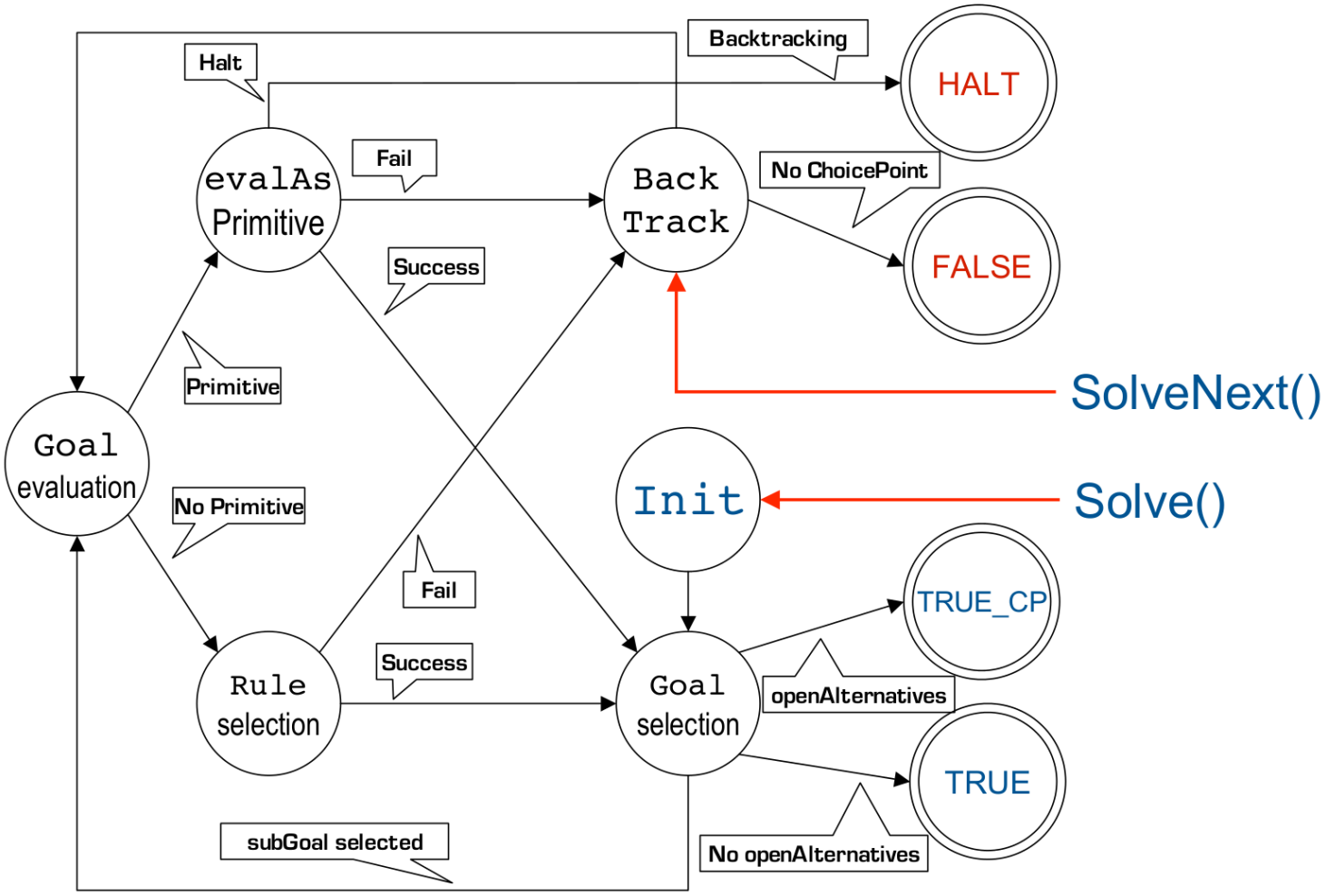

Figura: Macchina a stati alla base del motore TuProlog.

Risoluzione in TuProlog

- Quando l'engine considera una determinata clausola instanzia un `ExecutionContext` ad essa dedicato
- Ogni `ExecutionContext` è linkato ai precedenti mediante la variabile `fatherCtx`
- È necessario agire sulla dinamica del linking dei contesti per riuscire a scartare tutti i passaggi di risoluzione inutili.

L'implementazione attuale:

- Prevede l'attuazione dell'ottimizzazione all'interno dello stato *RuleSelection*, dopo la generazione dell'`ExecutionContext`
- Esegue il fetch sul `SubGoalStore` nello stato di *GoalSelection*
- A causa della dinamica del FSA non permette in nessun caso l'innesco della TRO

TRO in TuProlog, considerazioni

Volendo implementare una corretta procedura di TRO in TuProlog:

- sarà necessario agire sul meccanismo di linking degli `ExecutionContext` instanziati
- sarà probabilmente necessario integrare ulteriori informazioni all'interno delle strutture dati scambiate in fase di risoluzione
- bisognerà tenere bene a mente la dinamica del FSA per innescare l'ottimizzazione al momento opportuno

Conclusioni

- L'introduzione dell'indexing da parte Paolo Contessi e l'ottimizzazione del metodo `Struct.getTerm(int)` hanno permesso un miglioramento complessivo delle prestazioni dell'ordine del 20%.
- Le strutture dati attualmente utilizzate sembrano rappresentare il miglior trade-off tra tempi di accesso e overhead gestionale delle stesse.
- Non è stato possibile individuare ulteriori fattori dominanti per la caratterizzazione delle prestazioni.
- Un ulteriore step prestazionale potrebbe sicuramente essere compiuto riuscendo ad integrare efficacemente la *tail recursion optimization* all'interno dell'engine TuProlog.

Bibliografia

- Michele Damian
  Analisi di prestazione dell'interprete tuProlog su piattaforma Java
  Facoltà di ingegneria, Università di Bologna, 2008
- Paolo Contessi
  Relazione del progetto *Fast-Match*
  II facoltà di ingegneria, Università di Bologna, 2010
- Hassan Ait-Kaci
  Warren's Abstract Machine - A Tutorial Reconstruction
  MIT Press, febr. 1999
- A. Benini, A. Omicini, G. Piancastelli, A. Ricci
  The Architecture and Design of a Malleable Object-Oriented Prolog Engine
  DEIS, Università di Bologna, 2008
- *tuProlog Javadoc, source code*
  http://tuprolog.sourceforge.net
- *SICStus Prolog performance benchmarks*
  http://www.sics.se/isl/sicstuswww/site/performance.html