tuProlog Engine

Presentazione di

Enrico Siboni

Sistemi Autonomi – 2018/2019

1 . 1

Indice

In modalità slide premere ESC per la mappa

1 . 2

Introduzione

- **tuProlog** è un motore per l'esecuzione di programmi *Prolog*, realizzato con l'intento di essere *minimale*, altamente *configurabile* anche *dinamicamente*.
- Esso è stato realizzato in *Java* adottando dunque il **paradigma ad oggetti**.

2 . 1

Prolog

- Il **Prolog** è un linguaggio di programmazione che adotta il paradigma della programmazione logica
- Un programma Prolog è una sequenza di clausole di Horn nella forma `A :- B1, ..., Bn`
    - A è detta *testa*
    - `:-` simboleggia l'*implicazione logica*
    - `B1, ..., Bn` sono una congiunzione di termini, costituente il *corpo* della clausola
- L'esecuzione di un programma Prolog è comparabile alla dimostrazione di un teorema, mediante la regola di inferenza detta risoluzione

2.2

Risolutore

- Il cuore di *tuProlog* è il motore inferenziale di risoluzione costituito da una **macchina a stati finiti**
- Il processo di risoluzione adottatto è SLD (*Selective Linear Definite* resolution)
    - Questo processo definisce implicitamente un **albero di ricerca**, per le possibili computazioni alternative che possono dimostrare l'input sottomesso per la soluzione
        - L'albero definito è un'**or-tree**, cioè un albero in cui diversi rami indicano computazioni alternative
    - La risoluzione SLD non è deterministica, nel senso che non impone una strategia per la visita dell'albero di risoluzione, ma il Prolog definisce questa strategia in termini di una **Depth First Search**
    - Quando "la visita" incontra un nodo non deterministico, si memorizza il punto di scelta (**Choice Point**) in modo da poter esplorare ulteriori scenari (tamite **backtracking**) nel caso in cui la strada corrente porti a un fallimento

3 . 1

Database delle clausole

- Il database delle clausole contiene l'insieme delle clausole che formano la **teoria** (il programma) **Prolog** che potrà essere eseguito sottomettendo un input al motore inferenziale.
- Esso viene *consultato durante il processo di risoluzione* per determinare la veridicità di *fatti* applicando *regole*, al fine di dimostrare l'input iniziale.
    - Se si vuole vedere il risolutore come un *"Theorem Prover"*, che deve dimostrare la tesi iniziale data, si può vedere il database delle clausole come la raccolta di assiomi che costituiscono la **base di conoscenza** da cui partire per tentare la dimostrazione
- La consultazione rapida è coadiuvata da funzionalità di:
    - **indicizzazione**, effettuata fino al primo argomento della testa di ogni clausola
    - e **ricerca**, eseguibile per *indicatore* della testa (l'indicatore è una descrizione della "forma" di una struttura; es. `a(b, c)` ha indicatore `a/2`)

3 . 2

Database delle clausole: dettagli implementativi

- Le principali classi che realizzano il Database delle clausole sono: `ClauseDatabase`, `FamilyClauseList`, `FamilyClauseIndex` e `ClauseInfo`
    - `ClauseDatabase`: Realizza il database delle clausole vero e proprio, sotto forma di `HashMap`
    - `FamilyClauseList`: E' la stuttura dati "lista linkata" su cui *ClauseDatabase* si poggia, che realizza una momorizzazione indicizzata delle clausole tramite *FamilyClauseIndex*
    - `FamilyClauseIndex`: E' un indice Red-Black-Tree specializzato nella memorizzazione delle clausole
    - `ClauseInfo`: E' una classe che mantiene le informazioni fondamentali su ogni clausola

3 . 3

Librerie

- Le librerie sono teorie Prolog che contengono **funzionalità di base** utili alla costruzione di altri programmi Prolog
- In *tuProlog* ad esempio abbiamo di base le seguenti:
    - `BasicLibrary` che definisce predicati di base standard, ad eccezione di quelli di I/O
    - `IOLibrary` definisce i predicati di Input-Output descritti nello Standard di Prolog
    - `ISOLibrary` definisce i restanti predicati *Prolog Standard* non definiti nelle precedenti due librerie
    - `JavaLibrary` definisce predicati specifici per l'interazione *Prolog/Java*

3 . 4

Librerie: alcuni dettagli implementativi

- Le librerie in *tuProlog* sono oggetti che:
    - Dichiarano una *teoria*, in formato di Stringa
        - Quando esse vengono caricate, si effettua il parsing di quest'ultima
        - Gli operatori che vengono riconosciuti in questa fase di parsing, sono quelli caricati al momento
    - Possono dichiarare *primitive*, *funzioni* e *direttive*, sotto forma di metodi con determinati vincoli di nome, tipi di argomenti, e loro numero
- Se si vuole che il *"backtracking"* sia *possibile* in un *predicato di libreria*, si deve scriverlo nella Stringa che verrà riconosciuta come teoria
- A livello implementativo abbiamo la classe astratta `Library` che codifica le funzionalità di base per l'implementazione di tutte le altre librerie, inclusi i meccanismi di *reflection Java* utili al riconoscimento di *primitive, funzioni e direttive*

3 . 5

Primitive

- Per primitive si intendono delle funzionalità che sono *richiamabili da* un programma *Prolog* la cui esecuzione non è demandata al risolutore ma all'**esecuzione di codice nativo** nel linguaggio di origine del motore inferenziale.
- Le primitive *sopperiscono* all'**inefficienza** e **difficoltà** di implementazione di certe funzionalità, in termini di librerie Prolog inerentemenete dichiarative
    - Funzionalità che sono invece di facile realizzazione, in un linguaggio imperativo

3 . 6

Primitive: alcuni dettagli implementativi

- Le primitive in *tuProlog* sono metodi:
    - Dichiarate all'interno di una classe che implementa una *Libreria*
    - Richiamabili tramite il meccanismo della *Reflection di Java* secondo la convenzione per cui
        - Il tipo di ritorno è *booleano*
        - Il nome della funzione Java deve avere lo stesso nome del *predicato Prolog* seguito da un _ (Underscore) e l'arità dello stesso
        - Il numero di argomenti della funzione deve essere compatibile con l'arità dichiarata, e del tipo corretto (`Term`)
- Le primitive, così implementate, non possono prendere parte al processo di "backtracking"
- Un esempio: se volessi richiamare la primitiva `ciao(enrico)`
    - La corrispondente signature Java all'interno di una Libreria sarebbe: `boolean ciao_1(Term arg)` in cui `arg` verrebbe istanziato con un atomo `enrico`

3.7

Funzioni

- Le funzioni in *tuProlog* sono come le *primitive* con la seguente differenza:
    - Il tipo di ritorno è un *termine Prolog* (`Term`)
- Le funzioni vengono valutate prima di valutare un goal nella sua interezza, siccome trasformano più termini in uno solo, grazie al predicato `is/2`
- Un esempio: se volessi valorizzare una variabile con il risultato di una funzione di somma `is(N, sum(2,3))`
    - La corrispondente signature Java all'interno di una Libreria sarebbe: `Term sum_2(Term arg1, Term arg2)` in cui
        - `arg1` verrebbe istanziato con 2
        - `arg2` verrebbe istanziato con 3

3.8

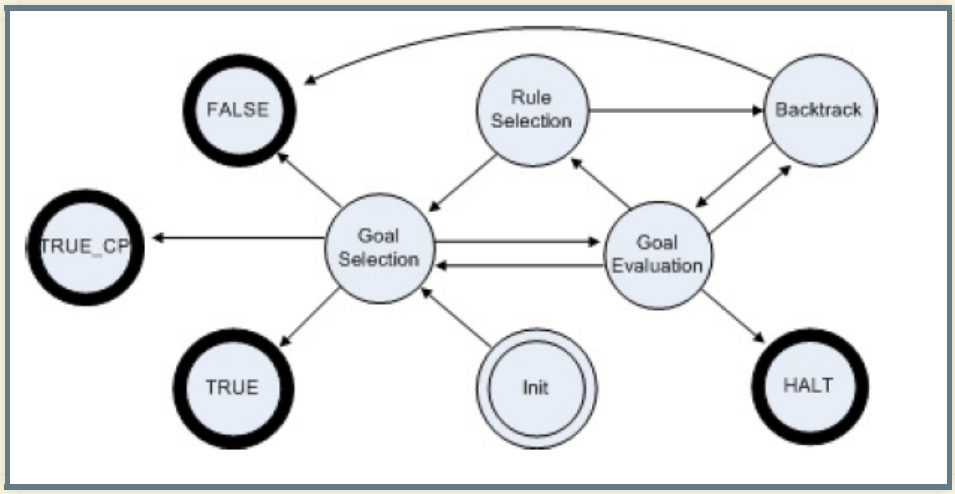
Direttive

- Le direttive in *tuProlog* sono come le *primitive* con la seguente differenza:
    - Il tipo di ritorno è *vuoto* (`void` in Java)
- Le direttive sono immediatamente *eseguite al caricamento* di una teoria *Prolog* all'interno del motore inferenziale
- Esse non possono essere composte, ogni direttiva può lanciare una sola *query*
- Ad esempio: se volessi lanciare la direttiva `ciao(enrico)` al caricamento della mia teoria
    - La corrispondente signature Java all'interno di una Libreria sarebbe: `void ciao_1(Term arg)` in cui `arg` verrebbe istanziato con un atomo `enrico`
    - Se la teoria contenesse la Stringa `:- ciao(enrico).`, la corrispondente direttiva verrebbe eseguita al caricamento

3.9

Macchina a stati finiti

- Il cuore di *tuProlog* è modellato come una **Macchina a Stati Finiti**, in questo modo è stato possibile separare e delineare le *diverse fasi* del processo di *risoluzione*, aprendo anche alla possibilità di inserirne di nuove



4 . 1

Strutture di appoggio

- Durante l'esecuzione del motore inferenziale viene fatto affidamento ad alcune strutture dati di supporto, tra cui:
    - **ExecutionContext** mantiene le informazioni cruciali riguardanti l'esecuzione corrente:
        - Quale è il *goal corrente*, quali sono le *variabili istanziate*, quale è il *contesto padre* di quello corrente, se ci sono *alternative aperte* ancora da valutare, ecc
    - **ChoicePointStore** che mantiene le informazioni relative a *quali* sono i punti di scelta, su cui ancora ci sono scelte non valutate
    - **SubGoalStore** che mantiene le informazioni relative ai sotto-goal che dovranno essere eseguiti

4.2

Stati finali

- Gli stati finali nella versione di tuProlog rappresentata nella figura precedente, sono 4, e li si raggiunge in questi modi:
    - **True**, se la risoluzione ha eseguito con successo l'input senza ulteriori alternative esplorabili
    - **True & Choice Point**, se la risoluzione ha eseguito con successo l'input e ci sono altre alternative di esecuzione aperte
    - **False**, se l'esecuzione è fallita, non riuscendo a dimostrare l'input con le teorie presenti
    - **Halt**, se l'esecuzione è stata arrestata forzatamente

4 . 3

StateInit

- Lo stato iniziale è molto semplice
  - Estrae il primo sotto-goal da eseguire
  - Inizializza l'*ExecutionContext* preparandolo per la computazione
  - Poi transita sempre nello stato **StateGoalSelection**

4 . 4

StateGoalSelection

- Lo stato di “selezione del goal” ha il compito di selezionare e preparare per l’esecuzione il prossimo sotto-goal
- I suoi step possono essere riassunti in:

    1. Carica il prossimo sotto-goal da eseguire dal contesto corrente
    2. Se non è presente alcun goal da eseguire, risale di un livello la catena degli *ExecutionContext* e riesegue dal punto 1.
       2.1. Quando non è possibile risalire la catena perchè si è arrivati all’inizio
        - e ci sono *ChoicePoint aperti*, transita in **True & Choice Point**
        - e non ci sono *ChoicePoint*, transita in **True**
    3. Se il goal caricato non è ben formato, transita in **False**
    4. Se il goal è ben formato, esso viene preparato per l’esecuzione e si transita nello stato **StateGoalEvaluation**

4.5

StateGoalEvaluation

- Lo stato di “valutazione del goal” ha il compito di verificare se il goal corrente è una primitiva e in tal caso eseguirla, gestendo le situazioni di errore relative
- Gli step sono:

    1. Controlla se il goal corrente, è una primitiva
    2. Se non è una *primitiva* transita immediatamente in **StateRuleSelection**
    3. Se il goal corrente è *una primitiva*, la esegue e possono verificarsi quattro situazioni:
       3.1. Essa ha *successo*, e si transita nello stato **StateGoalSelection**
       3.2. Essa *fallisce*, e si transita nello stato **StateBacktrack**
       3.3. Essa va in *errore* prima di terminare
        - Lanciando una eccezione di tipo *Halt*, e si transita nello stato **Halt**
        - Lanciando un’altro tipo di eccezione, e si transita nello stato **StateException**

4 . 6

StateRuleSelection

- Lo stato di “selezione delle regole” varifica se, all’interno del contesto corrente, è caricata una regola la cui testa *unifica* con il goal corrente; in caso affermativo predispone l’esecuzione del corpo della regola, altrimenti va in backtracking

1. Carica le possibili alternative di esecuzione del contesto corrente
    - Se vengo da *Backtracking*, stavo valutando già diverse alternative, le riprende
    - Altrimenti carica dal contesto corrente, le possibili alternative di esecuzione
        - Nel caso non ce ne siano, transita in **StateBacktrack**
2. Seleziona la prossima alternativa, evitando quelle già valutate, e crea il nuovo *ExecutionContext*
3. Rimpiazza le variabili presenti nella regola caricata, per non interferire con nomi di altre già istanziate
4. Predispone le strutture dati interne per ricordare questo *ChoicePoint*, oppure lo elimina se si è consumata l’ultima alternativa possibile
5. *Unifica* il goal corrente con la testa della regola, instanziando le variabili
6. Imposta il contesto corrente su quello appena creato, e transita in **StateGoalSelection**

4 . 7

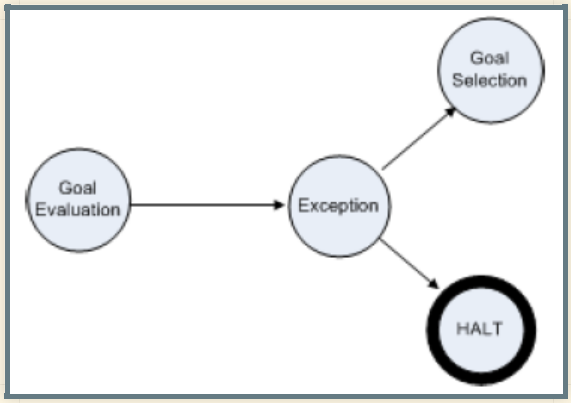
StateBacktrack

- Lo stato di “backtracking” (tradotto “di marcia indietro”), si occupa di annullare le modifiche intercorse tra il contesto corrente e l’ultimo *ChoicePoint* con percorsi aperti, permettendo quindi di provarne uno alternativo
- Gli step sono:
    1. Carica la prossima scelta selezionabile, nel più recente *ChoicePoint* generato
        - Se non ho ulteriori scelte percorribili, transito in **False**
    2. Ricarica il goal al momento della scelta, e elimina le modifiche effettuate sul contesto fino a quel momento
    3. Ripristina la coerenza nei contesti di esecuzione, padri di quello corrente, rimuovendo le assegnazioni di variabili relative all’ultima scelta
        - Se durante le operazioni di ripristino, un goal non risultasse ben formato, transita in **False**
        - Se tutto va a buon fine, transita in **StateGoalEvaluation**

4 . 8

StateException 1...

- Lo stato di “eccezione” tratta quelle situazioni in cui si è verificato un errore, durante l’esecuzione, e si deve in qualche modo gestirlo, ripristinando l’operatività
- Esso non era previsto nella definizione iniziale della macchina a stati, ma lo Standard Prolog lo richiede; in figura vediamo come si allaccia agli altri stati:



4 . 9

StateException ...2

- Gli step dello stato di “eccezione” sono:

    1. Estrae il termine descrittivo dell’errore, dal goal corrente
    2. Controlla se il contesto corrente ha un padre
       2.1. In caso negativo, nessuno può gestire l’errore, transita in **Halt**
    3. Controlla se il goal del contesto padre, può gestire l’errore
       3.1. In caso negativo, imposta come contesto corrente il padre, e riparte dal punto 2.
    4. Imposta come contesto corrente il padre, ed *elimina* tutti i *ChoicePoint* generati, a partire dal contesto corrente, fino al contesto che ha lanciato l’errore
    5. *Unifica* il termine descrittivo dell’errore con il *gestore dell’errore (catch)*, per propagare l’informazione
    6. Prepara per l’esecuzione il “goal di recupero” dall’errore
       6.1. Se esso non è ben formato, transita in **False**
    7. Transita in **StateGoalSelection**

4 . 10

Simulazione di esecuzione: Esempi

- Prendiamo, come esempio di base, la teoria *Prolog* seguente:

```
f(A) :- p(A).
g(B) :- p(B), !.
h(C) :- p(C), throw(my_error).

p(a).
p(b).
```

- Analizzeremo passo passo l'esecuzione dei seguenti goal, dati in input al motore inferenziale:
  - `f(Var).`
  - `g(Var).`
  - `h(Var).`

5 . 1

Esecuzione di `f(Var).`

1. *StateInit*: inizializza un contesto vuoto, con `f(Var)` all'interno di un `SubGoalStore`
2. *StateGoalSelection*: carica `f(Var)` dal `SubGoalStore`
3. *StateGoalEvaluation*: non essendo una primitiva non fa nulla
4. *StateRuleSelection*:
   4.1. Carica la regola `f(A) :- p(A)`
   4.2. Rinomina le variabili `f(A_1) :- p(A_1)`
   4.3. Imposta `A_1` come sostituzione di `Var`, dopo l'unificazione
   4.4. Il nuovo contesto corrente ha `p(A_1)` all'interno del proprio `SubGoalStore`
5. *StateGoalSelection*: carica `p(A_1)`; *SatateGoalEvaluation*: non fa nulla come sopra
6. *StateRuleSelection*:
   6.1. Carica i fatti `p(a)` e `p(b)`
   6.2. Crea un *ChoicePoint* per ricordre che sarà fatta una scelta, ripercorribile
   6.3. Sceglie `p(a)` e imposta `a` come sostituzione di `A_1`
   6.4. Il nuovo contesto corrente ha un `SubGoalStore` vuoto
7. *StateGoalSelection*: non ha ulteriori sotto-goal da eseguire, ma c'è un *ChoicePoint*
8. *True & ChoicePoint*: `Var -> a`; l'utente può decidere di proseguire l'esecuzione (con le prossime alternative) o terminarla

5.2

Esecuzione di `g(Var).`

- Ripercorre l'esecuzione di `f(Var)` fino al punto 5 compreso, utilizzando la regola `g(B) :- p(B), !`, poi:

6. *StateRuleSelection*:

   6.1. Carica i fatti `p(a)` e `p(b)`
   6.2. Crea un *ChoicePoint* per ricordre che sarà fatta una scelta, ripercorribile
   6.3. Sceglie `p(a)` e imposta `a` come sostituzione di `B_1`

7. *StateGoalSelection*: carica `!`, che ha alias `cut`
8. *StateGoalEvaluation*: esegue la primitiva relativa (`cut_0`) che elimina i *ChoicePoint* aperti relativi alla regola in cui viene eseguito
9. *StateGoalSelection*: non ha ulteriori sotto-goal da eseguire, e nessun *ChoicePoint*
10. *True*: `Var -> a`; questa è l'unica soluzione data in output siccome le alternative sono state tagliate

5.3

Esecuzione di `h(Var).`

- Ripercorre l'esecuzione di `f(Var)` fino al punto 5 compreso, utilizzando la regola `h(C) :- p(C), throw(my_error)`, poi:

6. *StateRuleSelection*:

   6.1. Carica i fatti `p(a)` e `p(b)`
   6.2. Crea un *ChoicePoint* per ricordre che sarà fatta una scelta, ripercorribile
   6.3. Sceglie `p(a)` e imposta `a` come sostituzione di `C_1`

7. *StateGoalSelection*: carica `throw(my_error)`
8. *StateGoalEvaluation*: esegue la primitiva relativa (`throw_1`), la quale lancia l'eccezione `PrologError` con il termine `my_error`
9. *StateException*: nessun padre ha un goal di tipo `catch/3` che possa gestire l'errore
10. *Halt*

5 . 4

Problemi dell’attuale implementazione

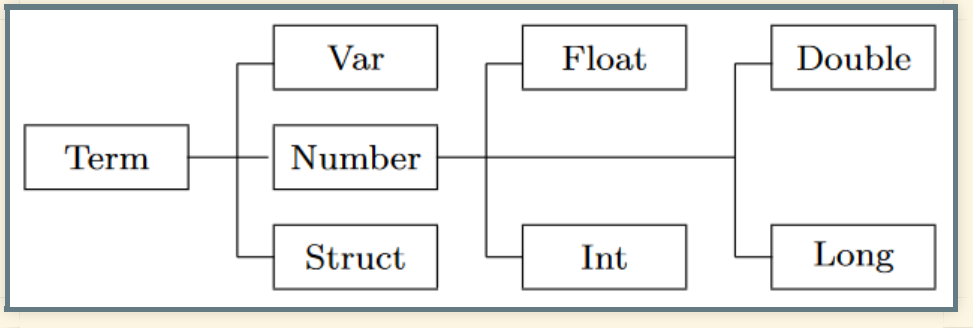
L’attuale implemetazione ha diversi problemi; alcuni dovuti a interventi ripetuti nel tempo, che hanno modificato il comportamento di varie parti, perdendo di vista l’insieme; altri causati dai limiti della tecnologia al momento della realizzazione.

- Tra i principali:
    - Classi di modello del dominio con errato livello di dettaglio
    - Classi di modello mutabili
    - Gestione interna di multipli flussi di controllo senza possibilità di intervento su questi parametri

6 . 1

Problema: Classi di modello del dominio con errato livello di dettaglio

Nell'attuale implementazione la gerarchia delle classi del dominio si limita alla seguente:

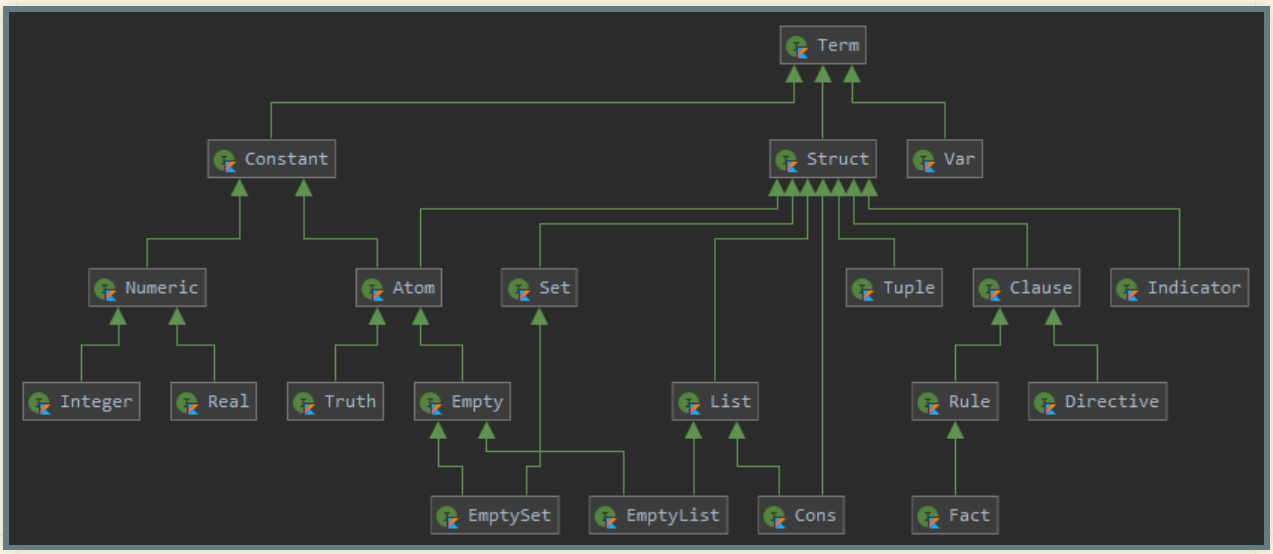


- La suddivisione dei termini tra Strutture, Numeri e Variabili non cattura la variabilità che si può avere al di sotto di `Struct`: (Atomi, Clausole, Liste, ecc.), perdendo anche la possibilità di fare un type checking a livello di linguaggio di programmazione
- Inoltre il dettaglio con cui si distinguono le possibili istanze di `Number`, risulta praticamente inutile

6 . 2

Soluzione: Classi di modello del dominio con errato livello di dettaglio

Si propone una rivisitazione della gerarchia delle classi di dominio, come segue:



- In questa gerarchia viene dato ampio spazio alle specizlizzazioni di `Struct`, che occupano così il loro posto come tipi di “prima classe”

6 . 3

Problema: Classi di modello mutabili

Nell'attuale implementazione le classi di modello sono mutabili, ciò però rende il codice che le gestisce *meno sicuro* a meno che non siano fatte copie difensive, perchè ci si espone a possibili mutazioni non volute. Ormai è noto che **la mutabilità**, è da evitare [1] se non si stanno cercando le massime performance in un software, perchè lo complica.

- Nel caso specifico la mutabilità è stata inserita ad esempio:
  - Nelle *Variabili* per permettere loro di mantenere il riferimento alla loro sostituzione
  - Nel *contesto corrente* per permettere la memorizzazione di volta in volta del goal corrente e tutta un'altra serie di informazioni che possono mutare durante l'esecuzione
- Questo ha portato alla necessità (per supportare il "backtracking") di creare:
  - *Strutture dati ausiliarie* per mantenere la storia delle modifiche
  - *Procedure ausiliare* articolate e complesse per la navigazione delle suddette strutture e il ripristino di uno specifico stato precedente

1. Bloch, Joshua. Effective java. Addison-Wesley Professional, 2017, **Item 17**. ↩

1

Soluzione: Classi di modello mutabili

Per semplificare al massimo il codice sviluppato, e seguendo le buone pratiche, si propone di fare dell'**immutabilità** la condizione di default per tutte le classi.

- In questo modo è possibile guadagnare:
  - Una **maggiore semplicità** del codice di modellazione del dominio
  - **Oggetti condivisibili** senza problemi di accessi e modifiche indesiderate, eliminando alla radice la necessità di creare copie difensive
  - L'**eliminazione** dello **stato** di *"backtracking"* dalla macchina a stati, siccome non avendo modificato nulla, non è più necessario annullare alcuna modifica
  - La **rimozione** delle **sovrastrutture** per mantenere le modifiche storiche, e relative procedure di navigazione
- Tutto ciò al prezzo di avere un oggetto diverso in memoria per ogni modifica necessaria; prezzo che si compensa in parte con l'ultimo punto illustrato qui sopra

6 . 6

Problema: Gestione interna di multipli flussi di controllo senza possibilità di intervento su questi parametri

Nell'attuale implementazione il motore inferenziale gestisce internamente i propri *Thread di controllo*, senza dare la possibilità di modificare il comportamento di default.

Questo risulta un problema quando ad esempio si vorrebbe utilizzare *tuProlog* in sistemi che non hanno molta potenza di calcolo e memoria.

Soluzione: Gestione interna di multipli flussi di controllo senza possibilità di intervento su questi parametri

Si propone un approccio per cui, dall'esterno, sia possibile definire quali risorse di calcolo debbano essere utilizzate durante il processo di risoluzione.

Ad esempio si potrebbe passare come parametro di input al risolutore, oltre all'input vero e proprio, anche la strategia con cui si vorrebbe venissero eseguite le computazioni interne.

6 . 7

Bibliografia

- Piancastelli, Giulio, et al. “The architecture and design of a malleable object-oriented Prolog engine.” Proceedings of the 2008 ACM symposium on Applied computing. ACM, 2008.
- Denti, Enrico, Andrea Omicini, and Alessandro Ricci. “Multi-paradigm Java–Prolog integration in tuProlog.” Science of Computer Programming 57.2 (2005): 217-250.
- Wikipedia - https://it.wikipedia.org

7