ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DEIS

TESI DI LAUREA

in

Linguaggi e Modelli Computazionali LS

Refactoring dell'interprete tuProlog per il supporto delle eccezioni in ambiente .NET

CANDIDATO:

Antonio Danilo Santoro

RELATORE:

Chiar.mo Prof. Enrico Denti

Sessione III

Anno Accademico 2008/09

INDICE

Introduzione

tuProlog[1], sviluppato presso il Dipartimento di Elettronica Informatica e Sistemistica della Facoltà di Ingegneria dell'Università degli Studi di Bologna, è un interprete per il linguaggio Prolog scritto in Java. tuProlog risulta essere leggero e dinamicamente configurabile, per le applicazioni e le infrastrutture Internet.

Questa tesi si pone l'obiettivo di realizzare una versione di tuProlog per il Framework Microsoft .NET in grado di supportare il meccanismo linguistico che Prolog offre per la gestione delle eccezioni.

Questo lavoro riprende un processo di reingegnerizzazione di tuProlog iniziato nel 2007, nel quale è stata compiuta la conversione di tale applicazione verso l'ambiente .NET, utilizzando il linguaggio Csharp (C#) e il Framework 2.0. La sua implementazione tuttavia non è completa, in quanto non permette di gestire gli errori che si possono verificare durante l'esecuzione. Allo stato attuale si ricorre infatti al semplice fallimento del predicato interessato dall'errore e per evitare che ciò avvenga verrà progettata un'estensione in grado di supportare la gestione delle eccezioni. Questo meccanismo è specificato nello standard ISO/IEC 13211-1, il quale viene implementato in diversi interpreti Prolog tra i quali figura anche il tuProlog nella versione Java.

Lo standard prevede un meccanismo per interrompere l'esecuzione di un predicato Prolog e propagare l'errore ad un livello più alto del programma che sia in grado di catturarlo e gestirlo. Per fare ciò vengono introdotti due predicati: `throw/1` permette di lanciare un'eccezione all'interno di un predicato in caso di errore, mentre `catch/3` può essere utilizzato per prevenire la terminazione forzata di un programma Prolog causata da eventuali errori. Il primo predicato ha, di fatto, una semantica simile a quella del `throw`

[1] tuProlog Guide: http://tuprolog.alice.unibo.it/

di C#, mentre il secondo è assimilabile a un blocco `try/catch`. Lo standard, inoltre, classifica in modo molto preciso gli errori che si possono verificare durante l'esecuzione dei predicati, e descrive per ognuno di essi la sintassi dell'eccezione da lanciare.

Il motore inferenziale di tuProlog è realizzato da una macchina a stati finiti facilmente estendibile, nella quale ogni stato individua un momento dell'esecuzione. Lo stato *GoalEvaluation* rappresenta un momento critico poiché in esso avviene la valutazione dei predicati primitivi dalla quale possono verificarsi condizioni di errore. Il progetto dell'estensione dovrà dunque tenere conto di tale architettura, e a questo scopo sarà necessario introdurre un nuovo stato *Exception* in cui transitare in caso di eccezioni con lo scopo di gestirle. A questo proposito tutti i predicati delle varie librerie di tuProlog saranno rivisitati in modo da lanciare le opportune eccezioni in caso di errore, evitando così il semplice fallimento.

Il progetto della gestione delle eccezioni in tuProlog consiste di tre fasi:

- Modifica del motore inferenziale;
- Implementazione dei predicati throw/1 e catch/3;
- Modifica di tutti i predicati di libreria.

Il presente lavoro di tesi è pertanto strutturato nel modo seguente:

I. Il primo capitolo descrive le caratteristiche e la malleabilità dell'architettura di tuProlog ed il funzionamento della macchina a stati finiti che è alla base del motore.
II. Il secondo capitolo analizza lo standard ISO Prolog il quale specifica i requisiti di comportamento dei predicati `throw/1` e `catch/3` e propone una classificazione degli errori.
III. Il terzo capitolo spiega come il Framework microsoft .NET affronta la problematica del controllo e gestione delle eccezioni evidenziandone le differenze rispetto a Java.

IV. Il quarto capitolo descrive le modifiche che abilitano il motore al riconoscimento ed alla gestione delle eccezioni.

V. Il quinto capitolo illustra le scelte che riguardano l'implementazione dei predicati `throw/1` e catch/3 sia nel caso delle eccezioni lanciate dai predicati di libreria che in quelle lanciate dai predicati che utilizzano oggetti C#.

VI. Il sesto capitolo rivisita, in modo più dettagliato, tutti i predicati delle librerie di tuProlog al fine di capire quali siano le eccezioni da lanciare ed in quali circostanze ciò debba avvenire.

VII. Il settimo capitolo descrive il piano del collaudo, le verifiche di correttezza e le performance dell'estensione.

VIII. Nell'ottavo capitolo vengono redatte le conclusioni.

1 L'interprete tuProlog

1.1 Caratteristiche generali

tuProlog è un interprete per il linguaggio Prolog, dinamicamente configurabile, leggero e basato su C#. La configurabilità è ottenuta grazie ad un meccanismo che consente di caricare e scaricare predicati, funtori e operatori, sia staticamente che dinamicamente all'interno di un motore tuProlog, grazie al concetto di libreria. Tra le librerie, alcune sono incluse nella distribuzione standard di tuProlog, altre possono essere scritte dall'utente o dallo sviluppatore. L'integrazione con C# è completa e trasparente: dal lato Prolog è possibile creare, rappresentare ed utilizzare ogni entità C# come un termine Prolog grazie alla libreria `CsharpLibrary`; dall'altro, è possibile invocare ed usare un motore tuProlog come un qualsiasi oggetto C#. Questa interazione bidirezionale consente di utilizzare le librerie di C# direttamente da Prolog, arricchendo in questa maniera l'interprete tuProlog di nuovi componenti, e offrendo al contempo la possibilità di sfruttare più motori tuProlog dall'ambiente C# eventualmente configurati in maniera diversa.

1.2 Architettura

Il cuore dell'architettura[2] di tuProlog è una macchina virtuale Prolog minimale, disponibile sotto forma di un oggetto C# auto-contenuto che offre un'interfaccia molto semplice. Le parti rilevanti di questa macchina virtuale sono controllate da una serie di entità gestori, ognuna con una ben determinata responsabilità. Si ha quindi un gestore per il motore inferenziale, uno per le

[2] Giulio Piancastelli, Alex Benini, Andrea Omicini, Alessandro Ricci. The Architecture and Design of a Malleable Object-Oriented Prolog Engine. 23rd ACM Symposium on Applied Computing (SAC2008). 16-20 March 2008.

teorie, uno per le librerie e uno per i predicati primitivi; questa suddivisione permette ai vari sottosistemi dell'architettura di evolvere il più possibile in modo indipendente l'uno dall'altro. Tale complessità è però nascosta all'utente, il quale interagisce con il sistema attraverso un'interfaccia unificata che ne fornisce una visione "semplificata", in linea con i principi del pattern Façade. Il motore di base può poi essere arricchito di funzionalità attraverso il meccanismo delle librerie.

1.2.1 Strutture dati, motore, teorie, librerie

Tutti i tipi di dato Prolog sono rappresentati da oggetti C#: `Term` è la classe base dei termini Prolog, suddivisi in *atomi*, *termini composti* (classe `Struct`), *numeri* (classi `Int, Long, Float, Double`) e *variabili* (classe `Var`).

Le classi importanti che costituiscono le API dell'interprete tuProlog riguardano il motore, le teorie e le librerie:

- `Prolog` è la classe che rappresenta un motore tuProlog, e fornisce un'interfaccia minimale che consente all'utente di impostare e leggere la teoria che deve essere utilizzata per la dimostrazione. Consente di caricare e scaricare librerie, risolvere un goal, composto da un oggetto `Term` o da una rappresentazione testuale di un termine (oggetto `String`). Un motore tuProlog può essere istanziato per mezzo del costruttore di default con delle librerie standard predefinite, o altrimenti con un insieme di librerie fornito come argomento del costruttore;
- `Theory` è la classe che incapsula una teoria tuProlog. Una teoria è rappresentata da un testo contenente una serie di clausole e/o direttive, ognuna seguita da un '.' e da un "whitespace". E' possibile costruire istanze di questa classe sia a partire da una rappresentazione testuale fornita direttamente come stringa, sia da una lista di termini che definiscono le clausole Prolog;

- `SolveInfo` è la classe che rappresenta il risultato di una dimostrazione. Le istanze di tale classe vengono restituite dal metodo *solve* di un motore tuProlog. In particolare, un oggetto *SolveInfo* fornisce i servizi necessari a verificare il successo di una dimostrazione (proprietà *IsSuccess*), ad accedere al termine che rappresenta la soluzione di una query (proprietà *Solution*) e ad accedere alla lista delle variabili con i rispettivi legami.
- `Library` è la classe che rappresenta una libreria tuProlog. Un motore tuProlog può essere esteso dinamicamente tramite il caricamento e lo scaricamento di librerie. Ogni libreria è in grado di fornire un insieme specifico di predicati, funtori e teorie correlate in modo da definire nuovi operatori. Una libreria può essere caricata attraverso il metodo *loadLibrary* del motore tuProlog.

1.2.2 Librerie standard

Le librerie sono il mezzo attraverso il quale tuProlog realizza la sua caratteristica di configurabilità dinamica. Per scelta progettuale il motore è un nucleo minimale e leggero, esso include infatti solo pochi predicati "built-in" definiti staticamente al suo interno. Ogni altra forma di funzionalità, come predicati, funtori, flag ed operatori, viene fornita dalle librerie, le quali possono essere aggiunte o tolte dinamicamente in ogni istante dal motore. tuProlog include all'interno della sua distribuzione quattro librerie standard, le quali sono caricate automaticamente dal costruttore di default nel motore. Le librerie che forniscono le caratteristiche e le funzionalità di base al motore Prolog sono:

- `BasicLibrary` fornisce i predicati, i funtori e gli operatori di base di Prolog necessari alla creazione, unificazione e decomposizione di termini Prolog, al confronto di termini ed espressioni, alla creazione e distruzione dinamica di clausole, ad impostare, aggiungere e leggere teorie ed alla valutazione di espressioni;

- `DCGLibrary` fornisce il supporto per la “Definite Clause Grammar”, la quale è un'estensione delle grammatiche “context free” ed è usata per descrivere linguaggi naturali e formali. Questa libreria non viene caricata automaticamente quando viene creato un motore tuProlog;
- `IOLibrary` fornisce alcuni predicati di base per l'I/O con, ad esempio, l'obiettivo di leggere un termine Prolog da uno stream di input o di scriverne uno su uno stream di output;
- `ISOLibrary` fornisce predicati e funtori che fanno parte della sezione built-in nello standard ISO non forniti dalle precedenti librerie. Tra questi vi sono, ad esempio, dei predicati per il controllo dei tipi, per la gestione e l'elaborazione degli atomi e dei funtori e per svolgere operazioni matematiche;
- `CsharpLibrary` fornisce predicati e funtori che consentono di creare, accedere ed utilizzare le risorse C# (come classi ed oggetti) come ad esempio i predicati che consentono di istanziare oggetti di una certa classe e di invocare metodi su di essi, ed i predicati per la gestione di array.

1.3 Il motore inferenziale

Il motore inferenziale di tuProlog è realizzato da una macchina a stati finiti facilmente estendibile, nella quale ogni stato individua un momento dell’esecuzione. Lo stato *GoalEvaluation* rappresenta un momento critico poiché in esso avviene la valutazione dei predicati primitivi dalla quale possono verificarsi condizioni di errore. Il progetto dell’estensione dovrà dunque tenere conto di tale architettura, e a questo scopo sarà necessario introdurre un nuovo stato *Exception* in cui transitare in caso di eccezioni con lo scopo di gestirle. A questo proposito tutti i predicati delle varie librerie di tuProlog saranno rivisitati in modo da lanciare le opportune eccezioni in caso di errore, evitando così il semplice fallimento.

La proprietà di estensibilità del tuProlog offre la possibilità di aggiungere facilmente (grazie all'utilizzo del pattern State) nuovi stati alla macchina, a fronte di nuovi requisiti di funzionamento. Questa caratteristica tornerà molto utile per poter estendere il motore tuProlog in modo da gestire le eccezioni.

1.3.1 L'automa esecutore

Il funzionamento della macchina a stati finiti è mostrato in Figura 1 e si compone dei seguenti stati:

- uno stato iniziale, che rappresenta l'inizio della procedura di risoluzione di una query;
- quattro stati intermedi, che rappresentano le attività eseguite durante il processo di risoluzione;
- quattro stati finali, che rappresentano i differenti risultati dell'esecuzione di una dimostrazione.

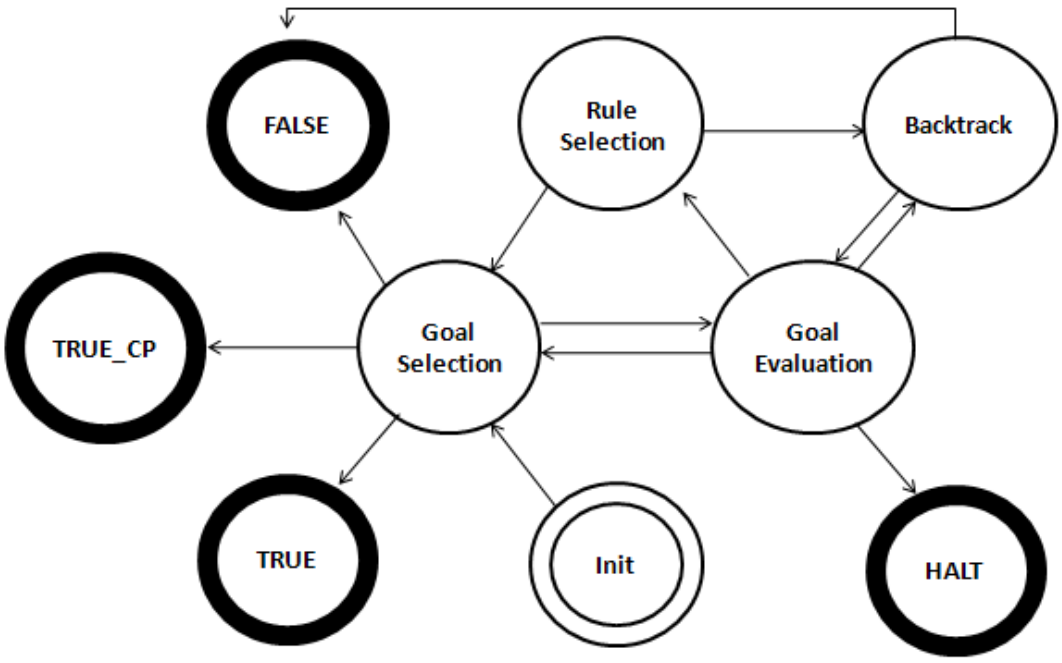

Figura 1 Automa a stati finiti che implementa il motore inferenziale.

1.3.2 Funzionamento dell'automa

Lo stato iniziale della macchina a stati finiti all'inizio di una dimostrazione è *Init*. Questo stato si occupa di inizializzare il motore tuProlog estraendo dalla query i subgoal da valutare ed inoltre costruisce un oggetto rappresentante il *contesto di esecuzione* del subgoal attuale per poi transitare nello stato successivo.

Lo stato *GoalSelection* si occupa di recuperare il prossimo goal da valutare dalla lista dei subgoal. In assenza di subgoal viene verificata l'esistenza dei punti di scelta aperti; se esistono la macchina termina nello stato *TRUE_CP*, dal quale può essere richiesta una ulteriore soluzione alla query iniziale, altrimenti la macchina si porta nello stato *TRUE* e il processo di risoluzione termina. In presenza di un subgoal è necessaria la verifica dell'effettiva valutabilità: se il subgoal è un numero la dimostrazione fallisce e il motore si porta nello stato *FALSE*, altrimenti in presenza di una variabile vengono eseguite una serie di operazioni che permettono alla dimostrazione di proseguire correttamente. La macchina transita poi nello stato di valutazione del goal.

Lo stato *GoalEvaluation* ha l'obiettivo di valutare un singolo subgoal che è stato precedentemente estratto dalla lista. In presenza di un predicato primitivo, la macchina si porta nello stato *GoalSelection* o *Backtrack*, in base al risultato della valutazione del predicato (successo o fallimento). Se invece il funtore principale del subgoal non rappresenta un predicato primitivo, la computazione prosegue soltanto se viene recuperata una clausola compatibile con tale subgoal nella base di conoscenza del motore. Questa funzione è svolta dallo stato *RuleSelection*.

In caso di errori durante la valutazione del subgoal, la macchina termina la computazione nello stato *HALT*.

Nello stato *RuleSelection* si cercano delle regole compatibili con il subgoal corrente nella base di conoscenza del motore. In assenza di regole compatibili,

la macchina si porta nello stato Backtrack ed inizia il backtracking. Altrimenti in caso di identificazione di una regola compatibile, si procede alla sua valutazione che consiste nella creazione di un nuovo contesto di esecuzione, nell'unificazione del subgoal con la testa della clausola e nell'aggiunta del corpo di questa alla lista dei risolventi mediante sostituzione del subgoal appena unificato. Se l'insieme di regole compatibili ha alternative aperte e non si proviene da un backtracking, viene creato in aggiunta un nuovo contesto per questo punto di scelta. In assenza di alternative aperte e provenendo da un backtracking, il contesto relativo a tale punto di scelta viene distrutto. Infine si esegue l'ottimizzazione della ricorsione tail sul contesto di esecuzione e la macchina si porta nello stato *GoalSelection*, per selezionare il prossimo subgoal da valutare.

Nello stato *Backtrack* si effettua un controllo sull'insieme delle clausole compatibili con il subgoal corrispondente all'ultimo punto di scelta aperto. In caso di insieme vuoto, la macchina transita nello stato *FALSE* e la dimostrazione fallisce. A questo punto le variabili e i risolventi vengono riportati al loro stato precedente, così come il contesto di esecuzione, e la macchina passa nello stato *GoalEvaluation* facendo ripartire il processo di risoluzione.

1.3.3 Implementare l'automa: il pattern State

La macchina a stati finiti che implementa il motore inferenziale di tuProlog è stata realizzata utilizzando il pattern State. Questo pattern permette ad un'entità di modificare il proprio comportamento quando il suo stato interno cambia. Nel caso di tuProlog, l'entità che cambia stato è il motore inferenziale `Engine`, gestito dall'`EngineManager`. Il motore può trovarsi in uno qualunque degli stati derivati dalla classe astratta `State`, e il comportamento del motore in ogni stato è definito dalla specifica implementazione del metodo astratto `doJob` della superclasse (Figura 2).

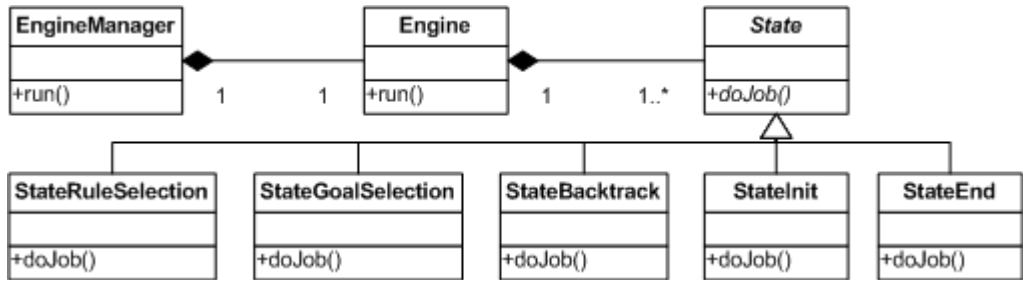

Figura 2 Diagramma delle classi del motore inferenziale implementato con il pattern State.

Dato che i diversi stati finali non esibiscono un particolare comportamento, essi sono modellati mediante un'unica classe `StateEnd`, il cui costruttore accetta un argomento intero che rappresenta lo specifico stato finale. Il pattern è implementato memorizzando nella classe `Engine` il prossimo stato in cui transitare.

2 Eccezioni nello standard ISO Prolog

2.1 Lo standard ISO Prolog

Lo standard ISO Prolog[3] (ISO/IEC 13211-1) è stato pubblicato nel 1995, con l'obiettivo di standardizzare il più possibile le diverse prassi adottate nelle numerose implementazioni Prolog.

Lo standard prevede un meccanismo per interrompere l'esecuzione di un predicato Prolog e propagare l'errore ad un livello più alto del programma che sia in grado di catturarlo e gestirlo. Per fare ciò vengono introdotti due predicati: `throw/1` permette di lanciare un'eccezione all'interno di un predicato in caso di errore, mentre `catch/3` può essere utilizzato per prevenire la terminazione forzata di un programma Prolog causata da eventuali errori. Il primo predicato ha, di fatto, una semantica simile a quella del `throw` di C#, mentre il secondo è assimilabile a un blocco `try/catch`. Lo standard, inoltre, classifica in modo molto preciso gli errori che si possono verificare durante l'esecuzione dei predicati, e descrive per ognuno di essi la sintassi dell'eccezione da lanciare. In sintesi lo standard introduce:

- i costrutti `catch/3` e `throw/1` per la gestione delle eccezioni
- classificazione degli errori

Vengono successivamente fornite delle suite di test per consentire agli sviluppatori di verificare la compatibilità con lo standard di una data implementazione Prolog.

[3] J.P.E. Hodgson. Prolog: The ISO Standard Documents. June 1999. http://pauillac.inria.fr/~deransar/prolog/docs.html

2.2 Errori ed eccezioni

Un errore è una circostanza particolare che interrompe il normale processo di esecuzione di un programma Prolog. Quando un sistema Prolog incontra una situazione di errore, lancia un'eccezione. Ad esempio, un'eccezione potrebbe essere lanciata da un predicato che individua un argomento non corretto.

Invece di terminare il programma come diretta conseguenza dell'errore, un meccanismo di gestione degli errori dovrebbe essere in grado di rilevare le situazioni di errore e, trasferire in modo controllato l'esecuzione ad un gestore, comunicandogli anche informazioni relative all'errore che si è verificato.

Il modello di gestione degli errori in Prolog segue due principi base:

- Il principio di confinamento dell'errore, che stabilisce di confinare un errore in modo da non propagarlo per l'intero programma. Un errore che si verifica in un componente deve essere catturabile ai suoi confini, invisibile all'esterno o riportato in maniera comprensibile. In Prolog ciò si ottiene per mezzo del predicato `catch/3;`
- Il principio del salto atomico, stabilisce che il meccanismo di gestione in caso di errore debba essere in grado di uscire in una sola operazione da un numero arbitrario di contesti di esecuzione annidati. In Prolog ciò si ottiene per mezzo del predicato `throw/1.`

2.2.1 Gestione dell'errore

Nel caso in cui si verifichi un errore durante l'esecuzione di un `Goal` è avviato il seguente meccanismo di gestione dell'errore:

a. Il subgoal in cui si è verificato l'errore viene sostituito dal subgoal `throw/1`;
b. Viene ricercata, nell'albero di risoluzione tra i nodi antenati, la più vicina clausola `catch/3` il cui secondo argomento unifica con l'argomento di `throw/1`;

c. Viene tagliato il percorso nell'albero di risoluzione che ha dato origine all'eccezione;
d. Viene rimosso il catcher, essendo valido soltanto per il goal protetto e non per il gestore;
e. Viene eseguito il gestore.

2.3 Il predicato ISO throw/1

Il predicato `throw(Error)` lancia l'eccezione `Error` che viene catturata dal più vicino antenato `catch(Goal, Catcher, Handler)` nell'albero di risoluzione il cui secondo argomento unifica con `Error`. In assenza di una clausola `catch/3` in grado di unificare, l'esecuzione fallisce. Il predicato può essere chiamato esplicitamente da un utente nel suo programma Prolog, oppure implicitamente nei vari predicati di libreria.

2.4 Il predicato ISO catch/3

Il predicato `catch(Goal, Catcher, Handler)` serve per proteggere l'esecuzione di `Goal` da eventuali eccezioni e per prevenire la terminazione in presenza di errori. Nel caso in cui non vengano lanciate eccezioni durante l'esecuzione di `Goal` il predicato si comporta come `call(Goal)`. Altrimenti durante tale esecuzione viene lanciata un'eccezione attraverso `throw/1` e il sistema provvede a tagliare tutti i punti di scelta generati da `Goal` provando ad unificare il `Catcher` con l'argomento di `throw/1`. Se l'unificazione ha successo, viene eseguito `call(Handler)` mantenendo le sostituzioni effettuate nell'unificazione. Se invece l'unificazione fallisce, il sistema continua a risalire l'albero di risoluzione in cerca di una clausola `catch/3` in grado di unificare. Se tale clausola non esiste, il predicato fallisce. L'esecuzione riprende infine con il subgoal successivo a `catch/3`. Ugualmente a quanto accade conseguentemente al fallimento di un predicato,

in caso di eccezioni, gli effetti collaterali eventualmente verificatisi durante l'esecuzione del `Goal` (come ad esempio le modifiche alla base di conoscenza del sistema) non vengono comunque distrutti.

Pertanto `catch/3` è vero se:

- `call(Goal)` è vero, oppure
- `call(Goal)` è interrotto da una chiamata a `throw/1` il cui argomento unifica con `Catcher`, e `call(Handler)` è vero.

E' necessario poi precisare altri due aspetti:

- se `Goal` è un predicato non-deterministico, attraverso il backtracking è possibile che venga rieseguito; tuttavia l'`Handler` viene eventualmente eseguito una sola volta, dato che in seguito a un'eccezione tutti i punti di scelta generati da `Goal` sono distrutti;
- l'esecuzione di `Goal` è protetta da `catch/3`, ma non lo è l'esecuzione dell'`Handler`, ne consegue che il gestore non è protetto.

2.5 Classificazione degli errori

Secondo lo standard ISO, quando si verifica un errore in un subgoal, questo viene rimpiazzato dal subgoal `throw(error(Error_term, Implementation_defined_term))` dove `Error_term` è un termine Prolog definito nello standard che fornisce informazioni sull'errore. `Implementation_defined_term` è un termine non standard caratteristico di ogni implementazione che può eventualmente essere omesso. Lo standard ISO classifica gli errori in base alla struttura del termine `Error_term`; tale classificazione è piatta, quindi il meccanismo di controllo degli errori può lavorare attraverso il pattern matching. Le classi individuate dallo standard sono dieci, e in seguito sono elencati i differenti tipi di errore che si possono presentare:

1. `instantiation_error`: Si ha un errore di istanziamento quando un argomento di un predicato o uno dei suoi componenti è una variabile, mentre è necessario che sia istanziato;
2. `type_error(ValidType, Culprit)`: Si ha un errore di tipo quando il tipo di dato di un argomento di un predicato o di uno dei suoi componenti non è corretto, e non è una variabile (in tal caso si ha un errore di istanziamento);
3. `domain_error(ValidDomain, Culprit)`: Si ha un errore di dominio quando il tipo di dato di un argomento è corretto, ma il valore è illegale (fuori dal dominio);
4. `existence_error(ObjectType, Culprit)`: Si ha un errore di esistenza quando un predicato cerca di accedere a un oggetto inesistente, ad esempio un file;
5. `permission_error(Operation, ObjectType, Culprit)`: Si ha un errore di permesso quando non è consentita l'esecuzione dell'operazione `Operation` sull'oggetto `Culprit` di tipo `ObjectType`;
6. `representation_error(Flag)`: Si ha un errore di rappresentazione quando durante l'esecuzione viene superato un limite definito dall'implementazione, come ad esempio la massima arità di un predicato;
7. `evaluation_error(Error)`: Si ha un errore di valutazione di un funtore quando produce un valore eccezionale.
8. `resource_error(Resource)`: Si ha un errore di risorsa quando il motore Prolog ha insufficienti risorse per completare l'esecuzione;
9. `syntax_error(Message)`: Si ha un errore di sintassi quando dei dati vengono letti da una sorgente esterna, ma hanno un formato non corretto o non possono essere processati;

10. `system_error`: Gli errori di sistema infine sono errori inaspettati riscontrati dal sistema operativo, che non rientrano in nessuna delle categorie precedenti.

3 Gestione delle eccezioni in ambiente Microsoft .NET

La maniera in cui vengono gestiti gli errori, è stata migliorata nel Framework .NET e il meccanismo di C# per la gestione degli errori, consente di poter fornire un diverso comportamento per ogni tipo di condizione di errore e di poter separare il codice che identifica gli errori da quello che li gestisce[4].

Ogni volta che si verifica un problema critico durante l'esecuzione di un programma, il Common Language Runtime (CLR) di .NET crea un oggetto `Exception` in cui vengono forniti i dettagli sull'errore. `Exception` è la classe base di tutte le classi di eccezione. Le categorie di eccezioni che derivano dalla classe `Exception` sono due: `SystemException` e `Application`. Tutti i tipi nello spazio dei nomi System derivano da `SystemException`, mentre le eccezioni definite dall'utente devono derivare da `ApplicationException` in modo da differenziare gli errori della fase di esecuzione da quelli dell'applicazione.

3.1 Confronto tra Java e C#

Analogamente a Java, quando si dispone di un codice che può causare un'eccezione, è opportuno inserire tale codice in un blocco `try`. Uno o più blocchi `catch` immediatamente successivi rendono possibile la gestione degli errori. È inoltre possibile utilizzare un blocco `finally` per il codice da eseguire, indipendentemente dal fatto che venga generata un'eccezione.

Quando si utilizzano più blocchi `catch`, le eccezioni rilevate devono essere inserite in ordine di genericità crescente, dato che viene eseguito solo il primo

[4] Christian Nagel, Bill Evjen, *C# 2008 - Guida per lo sviluppatore,* Milano, Hoepli

blocco `catch` corrispondente all'eccezione generata. A differenza del compilatore Java, il compilatore C# impone questa procedura. Ad esempio, durante un'operazione di lettura da un file, può verificarsi un'eccezione `FileNotFoundException` o `IOException`. In questo caso è opportuno inserire per primo il gestore `FileNotFoundException` più specifico.

A differenza di Java, C# non richiede necessariamente un argomento per un blocco `catch`. In mancanza di un argomento, il blocco `catch` viene applicato a qualsiasi classe `Exception`.

C# non supporta le eccezioni verificate. In Java le eccezioni di questo tipo vengono dichiarate utilizzando la parola chiave `throws` per specificare che un metodo può generare un particolare tipo di eccezione gestita dal codice chiamante.

3.2 Classi di eccezioni

In C#, un'eccezione è un oggetto creato quando si verifica una particolare condizione di errore. Questo oggetto contiene le informazioni che dovrebbero aiutare a tracciare il problema. Si possono creare le proprie classi di eccezioni o utilizzare classi di eccezioni predefinite.

Tutte le classi fanno parte del *namespace* `System`, con l'unica eccezione di `IOException` e le classi da essa derivate, che sono parte del *namespace* `System.IO`.

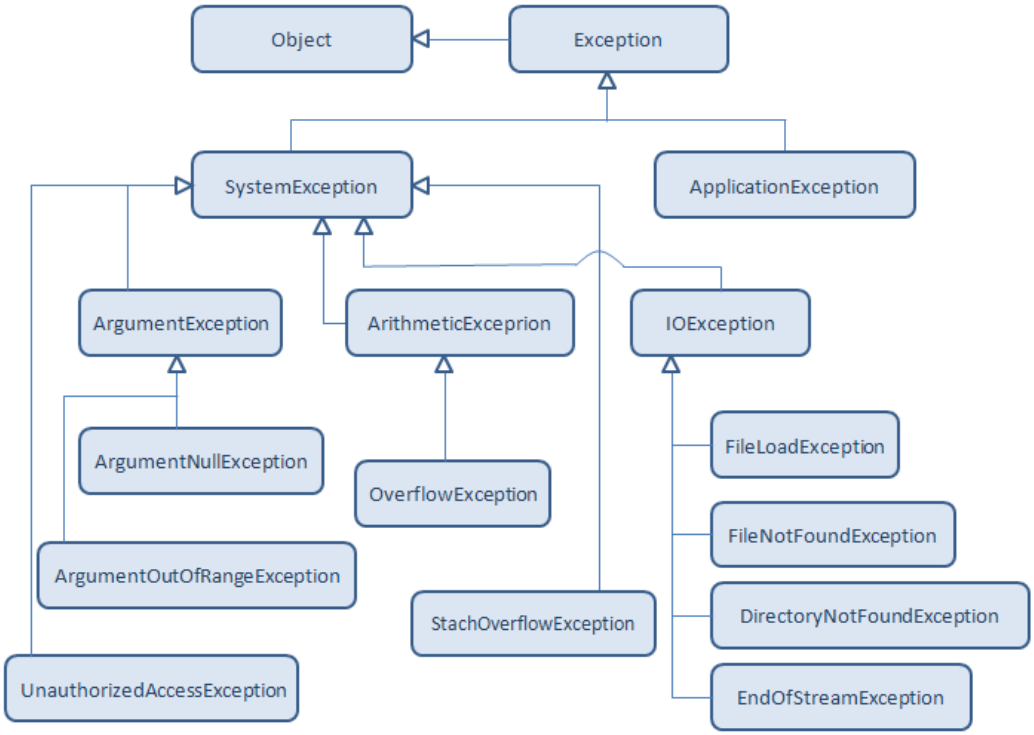

Figura 3 Gerarchia delle classi di eccezioni in .NET

La classe generica di eccezione, `System.Exception` è derivata da `System.Object`, ma solitamente non sono generati oggetti così generici poiché essi non forniscono alcuna specifica sulla condizione di errore. Le classi derivate da `System.Exception` sono:

- `System.SystemException`: questa classe è dedicata alle eccezioni lanciate dal runtime .NET e che solitamente sono lanciate dalla maggior parte delle applicazioni;
- `System.ApplicationException`: questa classe è importante perché è intesa come la base per qualsiasi classe di eccezione definita dallo sviluppatore. Pertanto, se venisse definita una qualsiasi eccezione che copre delle situazioni di errore specifiche della propria applicazione, (nel nostro caso CsharpException e PrologError) si dovrebbe derivarla direttamente o indirettamente da `System.ApplicationException`.

3.2.1 Caratteristiche

Le eccezioni supportano diverse caratteristiche. Innanzi tutto quando si verifica un'eccezione, il runtime rende disponibile un messaggio di testo per informare l'utente della natura dell'errore e per suggerire come risolvere il problema. Questo messaggio di testo è contenuto nella proprietà `Message` dell'oggetto eccezione. Durante la creazione dell'oggetto eccezione, è possibile passare una stringa di testo al costruttore per descrivere i dettagli di quella particolare eccezione. Se al costruttore non viene fornito alcun argomento di messaggio di errore, viene utilizzato il messaggio di errore predefinito. In secondo luogo la proprietà `StackTrace` riporta un'analisi dello stack che può essere utilizzata per determinare il punto del codice in cui si è verificato l'errore. Nell'analisi dello stack vengono elencati tutti i metodi chiamati e i numeri di riga nel file di origine dove vengono eseguite le chiamate.

Esistono inoltre proprietà in grado di identificare la posizione del codice, il tipo, il file della guida e il motivo dell'eccezione: `InnerException, HelpLink, HResult, Source, TargetSite e Data.`

3.1 Sollevare un'eccezione

Quando un errore viene rilevato in un blocco `try`, il codice solleva un eccezione, cioè istanzia una classe di oggetti eccezioni ad esempio:

```
throw new ArgumentException();
```

Non appena viene eseguita un'istruzione `throw` all'interno del blocco `try` viene individuato immediatamente un blocco `catch` associato al blocco `try` stesso. In presenza di più blocchi `catch`, è identificato il corretto blocco `catch` controllando a quale classe di eccezione è associato.

```
catch (ArgumentException ex)
{
```

```
throw new CsharpException(ex);
}
```

Il meccanismo della gestione delle interruzioni in C# è molto potente e flessibile in quanto è possibile avere istruzioni `throw` innestate in diverse chiamate di metodi all'interno di un blocco `try`. Quindi quando si incontra un'istruzione `throw` si ripercorrono immediatamente tutte le chiamate dei metodi nello stack, cercando la fine del blocco `try` e l'inizio dell'appropriato blocco `catch`.

3.2 Catturare le eccezioni – blocchi try/catch

Per far fronte alle possibili condizioni di errore del codice C#, si divide normalmente la parte rilevante del proprio programma sensibile ad una eventuale eccezione, in blocchi di tre tipi:

- i blocchi `try` incapsulano il codice che è parte delle normali operazioni del programma, e che può incappare in errori critici;
- i blocchi `catch` incapsulano il codice che avrà a che fare con le varie condizioni di errore che si potrebbero verificare nei rispettivi blocchi `try`;
- i blocchi `finally` incapsulano il codice che esegue ogni altra azione alla fine di un blocco `try` o `catch`.

3.2.1 Implementare più blocchi catch

E' possibile fornire tanti blocchi `catch` quanti sono gli specifici tipi di errore che si vogliono rilevare, tuttavia introdurre troppi blocchi `catch` può inficiare le prestazioni dell'applicazione.

Inizialmente vengono creati blocchi per trattare condizione di errore molto specifiche, per poi concludere con blocchi più generici che coprono tutti gli errori per i quali non sono presenti gestori specifici. Da ciò si evince che

l'ordine dei blocchi `catch` è importante, infatti scrivendo due blocchi con ordine di genericità invertita, si genera un errore in compilazione non essendo raggiungibile il secondo blocco `catch`.

A differenza di Java, C# implementa un blocco `catch` tra i più generici in assoluto:

```
catch
{
// gestisce l'eccezione
}
```

Lo scopo di questo blocco `catch` è quello di intercettare eccezioni sollevate da un altro codice non scritto in C#. Una caratteristica specifica del C# è rappresentata dal fatto che solo le istanze derivate da `System.Exception` possono essere sollevate come eccezioni; altri linguaggi possono non avere questa restrizione.

3.2.2 Blocchi Try/catch innestati

Un'altra caratteristica delle eccezioni è la possibilità di innestare blocchi `try` uno dentro l'altro come in questo caso:

```
try
{
      // punto A
      try
      {
      // punto B
      } catch {
      // punto C
      }
} catch {
```

```
// punto D
}
```

Un comportamento interessante si verifica se viene sollevata un'eccezione nel punto C. Se il programma è al punto C dovrebbe già processare un'eccezione sollevata al punto B. Sebbene sia del tutto legittimo sollevare un'altra eccezione all'interno del blocco `catch`, l'eccezione in questo caso viene trattata come se fosse generata dal blocco `try` esterno (come se fosse generata dal punto A) e quindi si cerca un gestore in un blocco `catch` esterno (ad esempio in D).

4 Refactoring di tuProlog

4.1 Analisi del Problema

Per abilitare il motore al riconoscimento e alla gestione degli errori in tuProlog deve essere progettata un'estensione della sua architettura. Analogamente all'approccio seguito nella versione Java di tuProlog[5], tale estensione può avvenire attraverso un'evoluzione del motore inferenziale descritto nel paragrafo 1.3, introducendo un nuovo stato nella macchina a stati finiti che ne è alla base. Infatti poiché nella conversione dalla versione Java di tuProlog a quella corrispondente su piattaforma .NET è stata preservata l'architettura dell'applicazione[6], è possibile abilitare il motore al supporto delle eccezioni seguendo lo stesso approccio.

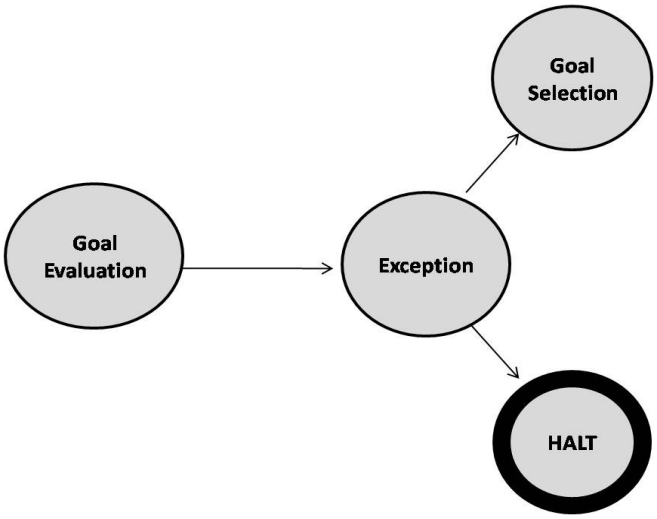

Figura 4 L'introduzione dello stato Exception per abilitare il motore alla gestione degli errori.

[5] Matteo Iuliani, *Estensione di un interprete Prolog per la gestione delle eccezioni*, 2009 tesi di laurea in Linguaggi e modelli computazionali, Università di Bologna, a.a 2008/2009.
[6] Mattia Generali, *Reingegnerizzazione del supporto per l'interprete tuProlog su piattaforma Microsoft .NET,* tesi di laurea in Linguaggi e modelli computazionali, Università di Bologna, a.a 2007/2008.

L'introduzione di un nuovo stato *Exception,* in grado di gestire le eccezioni, implica la modifica del comportamento dello stato *GoalEvaluation*. Grazie alla malleabilità architetturale con cui è stato progettato tuProlog è possibile estendere il motore modificando soltanto i due stati coinvolti nella transizione lasciando intatti tutti gli altri; ciò è possibile perché le transizioni tra i diversi stati sono gestite direttamente dagli stati stessi.

Infatti come mostrato in Figura 4, il nuovo stato è raggiungibile soltanto a partire dallo stato *GoalEvaluation* poiché è proprio in quest'ultimo che si può verificare un errore.

4.2 Il nuovo funzionamento dell'automa

Nello stato *GoalEvaluation,* durante la valutazione di un subgoal, può verificarsi un errore che fa transitare la macchina a stati finiti in un nuovo stato che prende il nome di *Exception*. Come specificato dallo standard, il subgoal in cui si è verificato l'errore viene sostituito dal subgoal `throw/1`, e la macchina si sposta nello stato *Exception*. In questo stato la macchina effettua una visita all'indietro dell'albero di risoluzione (composto di oggetti `ExecutionContext`) alla ricerca di un subgoal `catch/3` il cui secondo argomento unifica con l'argomento dell'eccezione lanciata. Durante tale ricerca, ogni `ExecutionContext` attraversato, deve essere potato in modo da non poter più essere eseguito o selezionato durante un backtracking.

Dopo aver identificato l'`ExecutionContext` corrispondente al corretto subgoal `catch/3`, la macchina inserisce il gestore dell'errore (`Handler`) in testa alla lista dei subgoal da eseguire, come definito dal terzo argomento di `catch/3`. E' necessario che il gestore mantenga le sostituzioni effettuate durante il processo di unificazione tra l'argomento di `throw/1` e il secondo argomento di `catch/3`. La macchina transita nello stato *GoalSelection* nel momento in cui viene individuato un nodo `catch/3` appropriato in modo da

poter gestire l'errore. Nel caso in cui non sia individuato un nodo, la macchina transita nello stato *HALT* determinando l'impossibilità di gestione dell'errore e la conseguente terminazione dell'esecuzione. Questa soluzione permette di rispettare il requisito fondamentale di correttezza che se una chiamata a un predicato non falliva prima dell'introduzione del meccanismo delle eccezioni, non deve fallire neanche ora e, viceversa, se non si è in grado di gestire l'errore, fallisce come prima.

La malleabilità architetturale di tuProlog consente l'evoluzione del suo motore inferenziale senza impattare sugli altri componenti dell'architettura: le modifiche restano infatti confinate all'interno della macchina a stati finiti.

4.3 Modifica della classe StateGoalEvaluation

Lo stato *GoalEvaluation* ha il compito di valutare il subgoal che è stato precedentemente estratto (nello stato *GoalSelection*) dalla lista dei subgoal.

Se il funtore principale del subgoal non rappresenta un predicato primitivo, la computazione può proseguire soltanto dopo l'individuazione, nella base di conoscenza del motore, di una clausola compatibile con tale subgoal. Successivamente il motore transita nello stato *RuleSelection*.

In presenza di un predicato primitivo, la macchina valuta il predicato e si può portare in quattro differenti stati:

1. Nello stato *GoalSelection,* se la valutazione del predicato ha successo;
2. Nello stato *Backtrack,* se la valutazione del predicato fallisce;
3. Nello stato *HALT,* se durante la valutazione del predicato si verificano errori gravi (non gestibili): il predicato in questione lancia una `HaltException` mediante il predicato built-in `halt/0` e il motore termina l'esecuzione;
4. Nello stato *Exception,* se il predicato lancia un'eccezione (gestibile).

I primi tre casi sono quelli previsti dall'attuale implementazione di tuProlog, mentre il quarto è quello che vogliamo aggiungere nel progetto di questa estensione.

Il comportamento dello stato *GoalEvaluation* è definito nel metodo `doJob()` della classe `StateGoalEvaluation`. In questo metodo la chiamata del predicato primitivo è inserita all'interno in un blocco `try` seguito da due blocchi `catch`. Se non si verificano errori durante la valutazione del predicato, si rimane all'interno del blocco `try`, mentre se viene lanciata dal predicato una `HaltException` essa viene catturata dal primo blocco `catch` che fa transitare il motore nello stato *HALT*. Il secondo blocco `catch` ha il compito di catturare un'istanza di `ApplicationException` ed ha lo scopo di gestire le eccezioni eventualmente lanciate dal predicato. La gestione delle eccezioni era stata quindi prevista dagli sviluppatori di tuProlog, ma non ancora implementata: attualmente nel `catch` viene semplicemente stampato lo `stacktrace` dell'errore (che all'atto pratico sarà un'istanza di `PrologError o CsharpException`) su `standardError` e, come nel caso precedente, il motore transita nello stato *HALT*.

Pertanto la modifica del comportamento dello stato *GoalEvaluation* per permettere la gestione delle eccezioni consiste nell'implementare correttamente il secondo blocco `catch`. Quindi, in base a quanto detto finora, nel blocco `catch`:

1. E' effettuato un cast dell'istanza di `ApplicationException` in un'istanza di `PrologError o CsharpException;`
2. Il subgoal in cui si è verificato l'errore viene sostituito dal subgoal `throw(Error)` o `csharp_throw(Exception)`, come specificato dallo standard ed in base al tipo di eccezione che è stata lanciata. La macchina ricava quindi le informazioni sull'errore dall'istanza di `PrologError` o `CsharpException` eccezione catturata, e assegna il

termine `throw/1 o csharp_throw/1` alla variabile `currentGoal` del corrente `ExecutionContext` del motore;
3. Il motore transita nello stato *Exception.*

4.4 Creazione della classe StateException

Analogamente alla versione Java di tuProlog la classe `StateException` modella il nuovo stato *Exception* ed ha il compito di gestire le eccezioni. Il comportamento è definito nel metodo `doJob()` della classe.

Per gestire le eccezioni, la macchina controlla il goal effettuando una visita all'indietro dell'albero di risoluzione (composto di oggetti `ExecutionContext`) alla ricerca di un subgoal `catch/3` il cui secondo argomento è unificabile con l'argomento dell'eccezione lanciata. Durante tale ricerca, ogni `ExecutionContext` attraversato viene potato, in modo da non poter più essere eseguito o selezionato durante un backtracking.

Una volta identificato l'`ExecutionContext` corrispondente al corretto subgoal `catch/3`, la macchina:

1. Taglia tutti i punti di scelta generati da `Goal` (attraverso il metodo `cut()` della classe `EngineManager`);
2. Unifica l'argomento di `throw/1` con il secondo argomento di `catch/3` (attraverso il metodo `unify()` della classe `Term`);
3. Inserisce il gestore dell'errore in testa alla lista dei subgoal da eseguire, mantenendo le sostituzioni effettuate durante il processo di unificazione (attraverso il metodo `pushSubGoal()` della classe `EngineManager`).

4.5 La classe EngineManager

Il motore inferenziale di tuProlog è realizzato attraverso il pattern State. L'entità che cambia stato durante l'esecuzione è il motore inferenziale `Engine`, tuttavia le istanze effettive dei diversi stati sono memorizzate nella classe `EngineManager` ovvero la classe che gestisce il motore inferenziale `Engine`. E' necessario, quindi, aggiungere un riferimento a un'istanza della classe che rappresenta il nuovo stato *Exception* alla classe `EngineManager`.

4.6 Le classi di eccezione

Le classi di eccezioni che vengono lanciate dai predicati di libreria in caso di errore, sono implementate in modo da incapsulare nel loro stato il tipo di errore verificatosi. Si individuano, come nel caso di tuProlog versione Java, due classi di eccezioni:

- `PrologError`: per catturare errori Prolog
- `CsharpException`: per catturare errori dei predicati della CsharpLibrary.

Mentre la classe `PrologError` , che segue lo standard ISO, è del tutto simile a quella della versione Java, la classe `JavaException` è sostituita da `CsharpException`.

4.6.1 PrologError

La classe `PrologError` è una classe che estende `ApplicationException` e cattura il concetto di errore Prolog. Viene lanciata un'istanza opportuna di PrologError ogni qual volta si verifica un errore durante l'esecuzione di un predicato, e il suo stato consiste nel termine Prolog che rappresenta l'argomento di `throw/1`. Inoltre, `PrologError` fornisce una serie di metodi di utilità chiamati dai predicati delle varie librerie per lanciare le corrette istanze corrispondenti all'errore che si è verificato. Secondo la classificazione dello standard ISO si possono lanciare le seguenti istanze:

1. `Instantiation_error`
2. `Type_error`
3. `Domain_error`
4. `Existence_error`
5. `Permission_error`
6. `Representation_error`
7. `Evaluation_error`
8. `Resource_error`
9. `Syntax_error`
10. `System_error`

4.6.2 CsharpException

A causa della similitudine nella gestione delle eccezione tra i linguaggi C# e Java discusse nel capitolo 3 il ruolo che ricopre la classe `CsharpException` è del tutto simile a quello della `JavaException`. Infatti la classe `CsharpException` è una classe che estende `ApplicationException` (e non più `Trowable` come nel caso Java) e rappresenta un errore che si è verificato durante l'esecuzione di un predicato della `CsharpLibrary`. Viene lanciata un'istanza opportuna di `CsharpException` ogni qual volta un predicato della `CsharpLibrary` rileva un errore durante la sua esecuzione, e il suo stato consiste nel nome della classe dell'eccezione C# corrispondente all'errore e nei suoi tre parametri che la caratterizzano (causa, messaggio associato e stacktrace). `CsharpException` ha la stessa funzione della classe `PrologError`, ma è utilizzata solo dai predicati della `CsharpLibrary` e non da tutte le altre librerie di tuProlog.

5 Implementare i predicati standard throw/1 e catch/3

5.1 Analisi del Problema

L'implementazione dei due predicati `throw/1` e `catch/3` risulta notevolmente facilitata, in quanto la maggior parte del lavoro relativo alla gestione delle eccezioni è svolto dalla macchina a stati finiti che implementa il motore inferenziale, e in particolar modo dallo stato *Exception*.

Tuttavia per decidere la giusta implementazione di questi predicati proposti dallo standard ISO è necessario tener presente che la generazione di eccezioni può essere determinata da cause differenti:

- Eccezioni Prolog generate all'interno dei predicati di libreria;
- Eccezioni C# generate all'interno di programmi Prolog che utilizzano oggetti C# durante la computazione.

Queste due differenti cause, producono eccezioni con caratteristiche differenti che vanno gestite in maniera distinta. Analogamente alla versione Java di tuProlog si è scelto, quindi, di progettare due coppie di predicati diversi in modo da trattare distintamente gli errori Prolog (attraverso i predicati `throw/1` e `catch/3`), dalle eccezioni C# (attraverso `csharp_throw/1` e `csharp_catch/3`). Questi ultimi predicati, infatti, permettono di avere una semantica più vicina a quella del mondo C# e sono utilizzati dal lato Prolog quando si usano oggetti C# attraverso i predicati della `CsharpLibrary`.

A seguire sono riportate anche le motivazioni che hanno influenzato le diverse implementazioni di questi predicati, cioè la possibilità di implementarli come predicati primitivi (C#) o come predicati di libreria Prolog.

5.2 Eccezioni Prolog nei predicati di libreria

Controllare la presenza di specifiche condizioni di errore all'interno dei predicati di libreria può essere particolarmente costoso in termini di prestazioni. Errori come quelli di istanziazione, di tipo o di dominio si riescono a diagnosticare prima dell'esecuzione di un predicato, mentre altre tipologie di errore, come quelli di esistenza o di valutazione, possono essere riscontrate solo durante l'esecuzione. Inoltre questa distinzione è valida sia per i predicati implementati in C#, sia per quelli implementati in Prolog. Dato che nell'architettura di tuProlog lo strato Prolog è costruito sopra lo strato C#, nel caso dei predicati di libreria implementati in C# il controllo può avvenire soltanto in C#.

5.2.1 Le guardie per i predicati implementati in Prolog

Nel caso dei predicati implementati in Prolog la presenza di errori può invece essere controllata sia dal lato Prolog sia dal lato C#, tuttavia la seconda alternativa è preferibile perché velocizza la risposta del motore.

Una possibile soluzione richiede di prevedere delle guardie apposite per i predicati Prolog, ovvero dei metodi C# da invocare prima dell'esecuzione di tali predicati allo scopo di controllarne i parametri, i permessi di esecuzione ed eventualmente lanciare le eccezioni appropriate. In questo modo la definizione dei predicati risulta divisa in due parti:

- la guardia espressa in C#;
- il corpo effettivo del predicato espresso in Prolog.

Nell'implementazione di questa soluzione va ricercato un giusto compromesso tra velocità di esecuzione e chiarezza, data la necessità di spezzare la definizione dei predicati in due parti.

5.2.2 Implementare throw/1

Il predicato `throw(Error)` è implementato in C# ed il suo compito è semplicemente quello di lanciare un'istanza di `PrologError` corrispondente all'errore `Error`:

```
public bool throw_1(Term error)
        {
            throw new PrologError(error);
        }
```

Come detto, tale istanza è catturata nello stato *GoalEvaluation* e le informazioni sull'errore verificatosi sono successivamente utilizzate dallo stato *Exception* per ricercare un opportuno subgoal `catch/3` in grado di gestirlo.

5.2.3 Implementare catch/3

Il predicato `catch(Goal, Catcher, Handler)` è invece implementato in Prolog poichè `Goal` potrebbe essere un predicato non-deterministico e quindi può essere rieseguito attraverso il meccanismo del backtracking:

```
catch(Goal, Catcher, Handler) :- call(Goal).
```

Il predicato si occupa semplicemente di chiamare `Goal`, mentre tutte le operazioni relative alla gestione degli eventuali errori che si possono verificare durante l'esecuzione di `Goal` vengono effettuate dallo stato *Exception*.

In definitiva `catch/3` è vero se:

- `call(Goal)` è vero, oppure
- `call(Goal)` è interrotto da una chiamata a `throw/1` il cui argomento unifica con `Catcher`, e `call(Handler)` è vero.

L'implementazione ideale dei predicati `throw/1` e `catch/3` richiede che essi siano il più vicino possibile al motore ovvero implementati come built-in e quindi residenti nella classe BuiltIn.

Tuttavia in tuProlog la libreria BuiltIn, differentemente dalle altre librerie, è gestita dal gestore dei predicati primitivi e non dal gestore delle librerie. Da ciò deriva un problema dato che il `PrimitiveManager` riesce a gestire soltanto predicati, funtori e direttive scritti in linguaggio nativo (C#) ma ignora i predicati scritti in Prolog, come `catch/3`.

Il `LibraryManager`, al contrario, riesce a gestire entrambi i tipi di predicati, pertanto una soluzione che ci permetta di inserire i due predicati tra i built-in del motore, potrebbe essere quella di far gestire anche `BuiltIn` dal `LibraryManager`. Tuttavia questa soluzione introduce un ulteriore problema dovuto alla definizione dell'operatore `:-` che è implementato nella `BasicLibrary`. L'operatore `:-` è indispensabile per effettuare correttamente il parsing della clausola `catch/3`, infatti se quest'ultima fosse implementata nella classe `BuiltIn` si introdurrebbe una teoria che usa operatori non ancora dichiarati causando quindi un errore di parsing.

Per queste ragioni si è scelto di collocare i due predicati nella `BasicLibrary` dove, comunque, sono situati predicati e funtori di base tipici di tutti i sistemi Prolog, come lo sono `throw/1` e `catch/3`.

5.3 Eccezioni C# nei programmi Prolog

tuProlog offre supporto per la programmazione multi-paradigma per cui programmi Prolog posso accedere ed utilizzare oggetti C# durante la computazione, questo implica anche la possibilità che vengano lanciate eccezioni C#.

La libreria `CSharpLibrary` ha lo scopo di consentire l'accesso e l'utilizzo di ogni componente e risorsa definita in C# direttamente da tuProlog .NET. In tutti i predicati della `CsharpLibrary` possono verificarsi errori durante l'esecuzione; tali errori possono essere dovuti all'invocazione di un predicato

con parametri non corretti oppure a condizioni eccezionali che si verificano durante l'esecuzione del predicato stesso.

Così come accade in tutte le altre librerie di tuProlog, questi errori provocano la terminazione del programma, mentre sarebbe opportuno trasferire in modo controllato l'esecuzione ad un gestore, comunicandogli le informazioni relative all'errore verificato. E' quindi necessaria una coppia di predicati, di cui uno permetta di lanciare un'eccezione e l'altro di gestirla.

Il principio di funzionamento è lo stesso dei due predicati standard `throw/1` e `catch/3`; tuttavia, per avere una semantica più vicina a quella del mondo C# si è preferito avere un diverso set di predicati per lanciare e gestire le eccezioni all'interno di programmi Prolog che utilizzano oggetti C# durante la computazione. Questi predicati sono `csharp_throw/1` e `csharp_catch/3`.

5.3.1 Corrispondenza tra la gestione delle eccezioni in C# e Prolog

Effettuando un paragone tra la gestione delle eccezioni in Prolog e in C#, si può notare che in Prolog non è presente il costrutto `finally` che troviamo invece in C# (e in Java). Consideriamo infatti il seguente codice C# relativo alla gestione di un'eccezione:

```
try {
     …     (1)
} catch (Exception e) {  (2)
     …     (3)
} finally {
     …     (4)
}
```

Se paragoniamo questa struttura a quella del predicato Prolog:

```
catch(Goal, Catcher, Handler)
```

possiamo stabilire una corrispondenza semantica tra:

- `(1)` e `Goal`
- `(2)` e `Catcher`
- `(3)` e `Handler`

mentre in `catch/3` non c'è nulla che corrisponda a `(4)`. Si può ovviare a questo considerando il blocco `finally` come una serie di operazioni che devono essere eseguite comunque, sia nel caso in cui ci siano eccezioni sia nel caso in cui non ce ne siano. Si può pertanto stabilire una corrispondenza più stretta tra i due linguaggi, considerando che i due pezzi di codice seguenti sono semanticamente equivalenti:

```
try {
     …    (1)
} catch (Exception e) {  (2)
     …    (3)
} finally {
     doSomething();
}
```

```
try {
     …    (1)
     doSomething();
} catch (Exception e) {  (2)
     …    (3)
     doSomething();
}
```

Le corrispondenze precedenti sono comunque valide, mentre `doSomething()` corrisponde ad un parte di codice Prolog presente sia in `Goal` che in `Handler`.

Un'altra differenza di C# è che ad ogni blocco `try` possono corrispondere più blocchi `catch`, visto che il meccanismo di cattura dell'eccezione lavora per *type matching* e la classificazione delle eccezioni è rigidamente gerarchica. Il secondo termine del predicato Prolog `catch/3` invece può catturare o un solo tipo di errore (se `Catcher` è parzialmente instanziato) o tutti gli errori

possibili (se `Catcher` è una variabile). Questa granularità potrebbe però non bastare se in un programma Prolog utilizziamo oggetti C# attraverso la `CsharpLibrary`.

Per preservare la semantica di C# si può quindi pensare, come nel caso di Java, all'implementazione di due predicati aggiuntivi da utilizzare proprio per lanciare e gestire le eccezioni C# in programmi Prolog: `csharp_throw/1` e `csharp_catch/3`.

5.3.2 Il predicato csharp_throw/1

Il predicato `csharp_throw/1` ha la forma:

```
csharp_throw(csharp_exception(Cause, Message, StackTrace))
```

dove l'atomo `csharp_exception` prende il nome della classe della specifica eccezione C# da lanciare espressa sotto forma di stringa (ad esempio `'ArgumentException'`), mentre i tre parametri rappresentano le tre parti che caratterizzano la tipica eccezione C#:

- `Cause` è una stringa che rappresenta la causa dell'eccezione, oppure `0` se la causa è inesistente o sconosciuta;
- `Message` è il messaggio associato all'errore (oppure `0` se il messaggio non c'è);
- `StackTrace` è una lista di stringhe, ognuna rappresentate uno stack frame (segmento logico dello stack).

Il predicato `csharp_throw(Exception)` lancia l'eccezione `Exception` che viene catturata dal più vicino antenato `csharp_catch/3` nell'albero di risoluzione il cui secondo argomento unifica con `Exception`. Se non viene trovata nessuna clausola `csharp_catch/3` in grado di unificare, l'esecuzione fallisce. Questo predicato è quindi utilizzato per segnalare un'eccezione che si verifica solo all'interno di un predicato della `CsharpLibrary`.

5.3.3 Implementare csharp_throw/1

Il predicato `csharp_throw(Exception)` non ha bisogno di essere implementato: si tratta infatti di un predicato fittizio che, assegnato alla variabile `currentGoal` del corrente `ExecutionContext` del motore, serve soltanto a trasferire informazioni sull'eccezione dallo stato *GoalEvaluation* allo stato *Exception*. Analogamente alla versione Java di tuProlog l'effettiva eccezione è incapsulata dall'istanza di `CsharpException (JavaException)` lanciata dai predicati della `CsharpLibrary` (`JavaLibrary`) e catturata nello stato *GoalEvaluation*.

5.3.4 Il predicato csharp_catch/3

Il predicato `csharp_catch/3` ha invece la forma:

```
csharp_catch(CsharpGoal, [(Catcher1, Handler1), … , (CatcherN, HandlerN)], Finally).
```

dove `Goal` è il goal da eseguire sotto la protezione dei gestori definiti dal secondo argomento, ognuno associato a un particolare tipo di eccezione C#. Il terzo argomento ha invece la semantica del `finally` di C# e rappresenta il predicato da eseguire alla fine di `Goal` o di uno dei gestori. È necessario comunque che il sistema al termine di `Goal` o di uno dei gestori metta in esecuzione `finally`, per poi proseguire normalmente con il predicato successivo a `csharp_catch/3`.

Più in dettaglio:

- `CsharpGoal` deve essere un predicato della `CsharpLibrary;`
- i catcher devono avere la forma `csharp_exception(Cause, Message, StackTrace);`
- nel caso in cui il `finally` non fosse necessario, si deve passare come terzo argomento un termine speciale (l'intero `0`).

Il predicato mette in esecuzione `CsharpGoal` e successivamente `Finally` se non vengono lanciate eccezioni durante l'esecuzione di `CsharpGoal`. Se invece durante tale esecuzione viene lanciata un'eccezione attraverso `csharp_throw/1`, il sistema provvede a tagliare tutti i punti di scelta generati da `CsharpGoal` (in caso di predicato non-deterministico come ad esempio `csharp_object_bt/3`) e prova ad unificare uno dei `catcher` con l'argomento di `csharp_throw/1`. Se l'unificazione ha successo, si esegue il gestore corrispondente al `catcher` che unifica con l'eccezione lanciata, mantenendo le sostituzioni effettuate nell'unificazione. Se l'unificazione invece fallisce, il sistema continua a risalire l'albero di risoluzione in cerca di una clausola `csharp_catch/3` che unifichi. Se tale clausola non esiste, il predicato fallisce. Al termine dell'esecuzione del gestore, il sistema mette in esecuzione `Finally`, per poi proseguire normalmente con il predicato successivo a `csharp _catch/3`. In caso di eccezioni, gli effetti collaterali che si sono eventualmente verificati durante l'esecuzione di `CsharpGoal` non vengono comunque distrutti.

`csharp _catch/3` è vero se:

- `CsharpGoal` e `Finally` sono veri, oppure
- `Call(CsharpGoal)` è interrotto da una chiamata a `csharp_throw/1` il cui argomento unifica con uno dei `catcher`, e ha successo sia l'esecuzione del gestore che quella di `Finally`

Bisogna poi precisare altri due aspetti:

- Se `CsharpGoal` è un predicato non-deterministico (es. `csharp_object_bt/3`), allora attraverso il backtracking è possibile che venga rieseguito; tuttavia in caso di eccezione viene eseguito un gestore e tutti i punti di scelta generati da `CsharpGoal` sono distrutti (quindi quel gestore o un altro gestore della lista non verranno più eseguiti);

- L'esecuzione di `CsharpGoal` è protetta da `csharp_catch/3`, mentre i gestori e `Finally` non sono protetti.

5.3.5 Implementare csharp_catch/3

Il predicato `csharp_catch(CsharpGoal, [(Catcher1, Handler1), …, (CatcherN, HandlerN)], Finally)` è implementato in Prolog perché `CsharpGoal` potrebbe essere un predicato non-deterministico (es. `csharp_object_bt/3`) e in tal caso deve essere rieseguito attraverso il meccanismo del backtracking.

```
csharp_catch(CsharpGoal, List, Finally) :-
call(CsharpGoal), call(Finally).
```

Il predicato chiama semplicemente `CsharpGoal` e `Finally` e successivamente tutte le operazioni relative alla gestione degli eventuali errori che si possono verificare durante l'esecuzione di `Goal` vengono effettuate dallo stato *Exception*. Trattandosi di un predicato da utilizzare nel contesto della `CsharpLibrary`, tale libreria è la sua collocazione più ovvia.

6 Refactoring dei predicati

Dopo aver modificato la macchina a stati finiti che è alla base del motore e dopo aver implementato i due predicati standard `throw/1` e `catch/3`, il passo successivo per estendere tuProlog in modo da renderlo capace di lanciare e gestire le eccezioni consiste nel modificare i vari predicati delle librerie comprese nella distribuzione, in modo che essi non falliscano in caso di errori, ma lancino le opportune eccezioni.

Per quel che riguarda il modo in cui è stato implementato il meccanismo di controllo degli errori, è utile distinguere i predicati espressi in C# da quelli espressi in Prolog. Nel primo caso le eccezioni vengono lanciate direttamente dai corrispondenti metodi C# ogni qualvolta si verifica un errore, mentre nel secondo caso sono lanciate da metodi "guardia" (sempre espressi in C#) invocati per controllare i parametri prima dell'esecuzione del predicato Prolog.

6.1 Modifiche ai predicati di libreria

E'quindi necessario analizzare il funzionamento dei vari predicati e considerare per ognuno di essi quali errori si possono verificare. I vari predicati lanciano così un'istanza di `PrologError` corrispondente all'eventuale errore: tale istanza è catturata nello stato *GoalEvaluation* del motore, mentre le informazioni sull'errore che si è verificato sono poi utilizzate dallo stato *Exception* per ricercare un subgoal `catch/3` opportuno che sia in grado di gestirlo.

6.1.1 Guardie C# per predicati Prolog

Come anticipato nel paragrafo 5.2.1, esistono errori che possono essere diagnosticati prima dell'esecuzione di un predicato (come gli errori di istanziazione, di tipo o di dominio), mentre altre tipologie di errore (come quelli di esistenza o di valutazione) sono tipicamente a riscontrabili solo durante l'esecuzione. Naturalmente è importante che il controllo della presenza degli errori avvenga sia per i predicati implementati in C#, sia per quelli implementati in Prolog.

Visto che nell'architettura di tuProlog lo strato Prolog è costruito sopra lo strato C#, per i predicati di libreria implementati in C# il controllo può avvenire soltanto in C#. Nel caso dei predicati implementati in Prolog la presenza di errori può essere invece controllata sia dal lato Prolog, sia dal lato C#. La seconda alternativa è sicuramente preferibile per velocizzare la risposta del motore.

Pertanto in ogni libreria sono previste delle guardie apposite per i predicati Prolog, ovvero dei metodi C# da invocare prima dell'esecuzione di tali predicati con lo scopo di controllare i parametri e i permessi di esecuzione ed eventualmente lanciare le eccezioni appropriate. Si ha quindi che la definizione dei predicati può risultare divisa in due parti: il corpo effettivo del predicato espresso in Prolog e la sua guardia espressa in C#.

Le guardie C#, implementate per il controllo dei soli predicati Prolog, sono elencate nelle librerie che vengono passate in rassegna nei paragrafi successivi. Poiché il meccanismo delle guardie è applicato agli stessi predicati di libreria della versione Java di tuProlog, si ha una corrispondenza biunivoca tra le implementazioni delle guardie.

6.1.2 BuiltIn

La libreria `BuiltIn` raggruppa i predicati che influiscono direttamente sul processo di risoluzione, quelli troppo basilari per poter essere definiti altrove e

quelli che per motivi di efficienza necessitano di essere definiti vicino al "core". Questi predicati sono sempre disponibili nel motore, e non possono essere mai scaricati. Come nella versione Java di tuProlog tutti i predicati della libreria BuiltIn sono implementati nei rispettivi linguaggi di programmazione e non in Prolog, per cui non sono previste guardie. Ognuno di questi predicati è stato rivisitato ed ora è in grado di lanciare i tipi di errore riportati in tabella 1 .

	Instantiation	Type	Domain	Existence	Evaluation
asserta/1	x	x			
assertz/1	x	x			
sretract/1	x	x			
abolish/1	x	x			
halt/1	x	x			
loadlibrary/1	x	x		x	
unloadlibrary/1	x	x		x	
scall/1	x	x			
is/2	x	x			x
stolist/2	x	x			
sfromlist/2	x	x			
sappend/2	x	x			
sfind/2	x	x			
get_prolog_flag/2	x	x	x		
set_prolog_flag/2	x	x	x		
sop/3	x	x	x		

Tabella 1 Riassunto dei predicati BuilIn e del tipo di eccezioni che ora lanciano.

Ad esempio il predicato:

asserta/1

`asserta(Clause)` è vero, con l'effetto collaterale che la clausola `Clause` è aggiunta all'inizio della base di conoscenza del motore.

La sua implementazione C# è:

```
public bool asserta_1(Term arg0)
{
      arg0 = arg0.GetTerm;
      if (arg0 is Struct)
      {
            _theoryManager.AssertA((Struct) arg0, true, null,false);
            return true;
      }
      if (arg0 is Var)
           throw PrologError.instantiation_error(_engineManager, 1);
      else
            throw PrologError.type_error(_engineManager, 1,
            "clause", arg0);
}
```

Le eccezioni che può lanciare sono:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))` se `Clause` è una variabile. `Goal` è il goal in cui si è verificato il problema (`asserta(Clause)`), `ArgNo` indica quale argomento del predicato ha causato il problema: in questo caso il primo (`1`)
- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `Clause` non è una struttura. `Goal` è il goal in cui si è verificato il problema (`asserta(Clause)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `Clause` (`clause`), `Culprit` è il termine errato trovato (`Clause`)

6.1.3 BasicLibrary

Questa libreria definisce alcuni predicati e funtori di base che si trovano di solito nei sistemi Prolog, con l'eccezione dei predicati di I/O. Alcuni predicati sono implementati in C# ed altri in Prolog e sono elencati nella tabella 2.

	Instantiation	Type	Domain	Evaluation	Syntax
textconcat/3	x	x			
num_atom/2		x	x		

set_theory/1	x	x			x
add_theory/1	x	x			x
agent/1	x	x			
agent/2	x	x			
***pred*(@expr, @expr)**	x	x		x	
***arg/3**	x	x	x		
***clause/2**	x				
***call/1**	x	x			
***member/2**		x			
***reverse/2**		x			
***element/3**		x			
***delete/3**		x			
assert/1	x	x			
***retract/1**	x	x			
***retractall/1**	x	x			
***findall/3**	x	x			
***setof/3**	x	x			
***bagof/3**	x	x			

Tabella 2 Riassunto dei predicati della libreria BasicLibrary e del tipo di eccezioni che ora lanciano. Con l'asterisco sono indicati i predicati per cui è prevista la guardia.

Inoltre sono implementati i seguenti predicati di guardia:

- **arg_guard/3**

Il predicato guardia `arg_guard/3` controlla i parametri del predicato di libreria `arg/3`, per fare ciò è stata modificata nella teoria della libreria `BasicLibrary` la clausola `arg/3` come segue:

```
arg(N,C,T):- C =.. [_|Args], element(N,Args,T).
```

in

```
arg(N,C,T):- arg_guard(N,C,T), C =.. [_|Args], element(N,Args,T).
```

La sua implementazione in C# è:

```
public bool arg_guard_3(Term arg0, Term arg1, Term arg2) {
            arg0 = arg0.GetTerm;
            arg1 = arg1.GetTerm;
            if (arg0 is Var)
                  throw
PrologError.instantiation_error(_engine.EngineManager, 1);
            if (arg1 is Var)
                  throw
PrologError.instantiation_error(_engine.EngineManager, 2);
            if (!(arg0 is Int))
                  throw
PrologError.type_error(_engine.EngineManager, 1,
                              "integer", arg0);
            if (!arg1.IsCompound)
                  throw
PrologError.type_error(_engine.EngineManager, 2,
                              "compound", arg1);
            Int arg0int = (Int) arg0;
            if (arg0int.IntValue < 1)
                  throw
PrologError.domain_error(_engine.EngineManager, 1,
                              "greater_than_zero", arg0);
            return true;
      }
```

Da cui il predicato `arg(N, Term, Arg)` è vero se `Arg` è l'`N` esimo argomento di `Term` (il conteggio parte da 1).

Per cui le eccezioni lanciate sono:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))` se `N` o `Term` sono variabili. `Goal` è il goal in cui si è verificato il problema (`arg_guard(N, Term, Arg)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`)
- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `N` non è un intero oppure `Term` non è un termine composto. `Goal` è il goal in cui si è verificato il problema (`arg_guard(N, Term, Arg)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `N` o `Term` (`integer` o `compound`), `Culprit` è il termine errato trovato (`N` o `Term`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)` se `N` è un intero minore di 1. `Goal` è il goal in cui si è verificato il problema (`arg_guard(N, Term, Arg)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidDomain` è il dominio previsto per `N` (`greater_than_zero`), `Culprit` è il termine errato trovato (`N`)

❖ **`clause_guard/2`** a guardia di `clause/2`

```
clause(H,B)                    :-      clause_guard(H,B),
L=[],'sfind'(H,L),member((':-'(H,B)),L).
```

Lancia eccezioni di istanziazione.

❖ **`call_guard/1`** a guardia di `call/1`

```
call(G) :- call_guard(G), 'scall'(G).
```

Lancia eccezioni di istanziazione e di tipo.

❖ **`all_solutions_predicates_guard/3`** a guardia `di setof/3, bagof/3, findall/3`

```
setof(Template,          Goal,          Instances)          :-
all_solutions_predicates_guard(Template, Goal, Instances), …

bagof(Template,          Goal,          Instances)          :-
all_solutions_predicates_guard(Template, Goal, Instances), …

findall(Template,          Goal,          Instances)          :-
all_solutions_predicates_guard(Template, Goal, Instances),…
```

Lancia eccezioni di istanziazione e di tipo.

❖ **`element_guard/3`** a guardia `di element/3`

```
element(P,L,E):- element_guard(P,L,E), element0(P,L,E).
```

Lancia eccezioni di tipo

❖ **`member_guard/2`** a guardia `di member/2`

```
member(E,L) :- member_guard(E,L), member0(E,L).
```

Lancia eccezioni di istanziazione e di tipo.

- **reverse_guard/2** a guardia `di reverse/2`

```
reverse(L1,L2):- reverse_guard(L1,L2), reverse0(L1,[],L2).
```

Lancia eccezioni di istanziazione e di tipo.

- **delete_guard/3** a guardia `di delete/3`

```
delete(E,S,D) :- delete_guard(E,S,D), delete0(E,S,D).
```

Lancia eccezioni di tipo.

- **retract_guard/1** a guardia `di retract/1 e retractall/1`

```
retract(Rule) :- retract_guard(Rule), …

retractall(Head) :- retract_guard(Head), …
```

Lancia eccezioni di istanziazione e di tipo.

6.1.4 ISOLibrary

Questa libreria definisce i predicati standard ISO non definiti nelle precedenti librerie e non appartenenti alla categoria dell'I/O.

	Instantiation	Type
atom_length/2	x	x
atom_chars/2		x
char_code/2		x
***sub_atom/5**		x

Tabella 3 Riassunto dei predicati della libreria ISOLibrary e del tipo di eccezioni che ora lanciano. Con l'asterisco sono indicati i predicati per cui è prevista la guardia.

Inoltre è implementato il seguente predicato di guardia:

- `sub_atom_guard/5` è a guardia di `sub_atom/5`

```
sub_atom(Atom,B,L,A,Sub):-          sub_atom_guard(Atom,B,L,A,Sub),
sub_atom(Atom,B,L,A,Sub).
```

Lancia eccezioni di tipo.

6.1.5 DCGLibrary

Questa libreria fornisce il supporto per le Definite Clause Grammar (DCG), che sono un'estensione delle grammatiche libere da contesto. Queste grammatiche sono molto utili per la descrizione dei linguaggi naturali e formali, e possono essere convenientemente espresse in Prolog.

	Instantiation
***phrase/2**	X
***phrase/3**	X

Tabella 4 Riassunto dei predicati della libreria DCGLibrary e del tipo di eccezioni che ora lanciano. Con l'asterisco sono indicati i predicati per cui è prevista la guardia.

Inoltre sono implementati i seguenti predicati di guardia:

- **phrase_guard/2** a guardia di `phrase/2`

```
phrase(C,L) :- phrase_guard(C,L), phrase0(C,L).
```

La sua implementazione C# è:

```
public bool phrase_guard_2(Term arg0, Term arg1) {
            arg0 = arg0.GetTerm;
            if (arg0 is Var)
                  throw
PrologError.instantiation_error(_engine.EngineManager, 1);
            return true;
            }
```

Lancia eccezioni di istanziazione.

- **phrase_guard/3** a guardia di `phrase/3`

```
phrase(C,L,R) :- phrase_guard(C,L,R), phrase0(C,L,R).
```

La sua implementazione è simile a `phase_guard/2`. Lancia eccezioni di istanziazione.

6.1.6 IOLibrary

Questa libreria fornisce alcuni dei predicati Prolog standard per permettere l'interazione tra i programmi Prolog e risorse esterne come file e canali di I/O.

	Instantiation	Type	Domain
see/1	x	x	x
tell/1	x	x	x
get0/1			
get/1			
put/1	x	x	
tab/1	x	x	
write/1	x	x	
print/1	x	x	
read/1			
nl/0			
text_from_file/2	x	x	
agent_file/1	x	x	
***solve_file/2**	x	x	
agent_file/1	x	x	

Tabella 5 Riassunto dei predicati della libreria IOLibrary e del tipo di eccezioni che ora lanciano. Con l'asterisco sono indicati i predicati per cui è prevista la guardia.

- **solve_file_goal_guard/2** a guardia di `solve_file/2`

```
solve_file(File,Goal)   :-   solve_file_goal_guard(File,Goal),
text_from_file(File,Text),text_term(Text,Goal),call(Goal).
```

La sua Implementazione in C#:

```
      public bool solve_file_goal_guard_2(Term arg0, Term arg1) {
            arg0 = arg0.GetTerm;
            arg1 = arg1.GetTerm;
            if (arg1 is Var)
                  throw
PrologError.instantiation_error(_engine.EngineManager, 2);
            if (!arg1.IsAtom && !arg1.IsCompound) {
```

```
                    throw
PrologError.type_error(_engine.EngineManager, 2,
                                "callable", arg1);
            }
            return true;
        }
```

`solve_file(TheoryFileName, G)` è vero se e solo se `TheoryFileName` è un file accessibile contenente una base di conoscenza Prolog, e come effetto collaterale risolve la query `G` in accordo a tale base di conoscenza.

Le eccezioni lanciate sono:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))` se `G` è una variabile. `Goal` è il goal in cui si è verificato il problema (`solve_file_goal_guard(TheoryFileName, G)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`)
- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `G` non è un goal invocabile. `Goal` è il goal in cui si è verificato il problema (`solve_file_goal_guard(TheoryFileName, G)`), `ArgNo` indica quale argomento ha causato il problema (`2`), `ValidType` è il tipo di dato previsto per `G` (`callable`), `Culprit` è il termine errato trovato (`G`)

Inoltre utilizza `text_from_file(File, Text)` con `TheoryFileName = File`, quindi lancia anche le sue stesse eccezioni.

6.2 CsharpLibrary

La libreria `CSharpLibrary` ha lo scopo di consentire l'accesso e l'utilizzo di ogni componente e risorsa definita in C# direttamente da tuProlog .NET. La maggior parte dei predicati presenti in questa libreria sfrutta il meccanismo della reflection. Così come per le altre librerie, anche i predicati della `CsharpLibrary` devono essere modificati in modo da non fallire in caso di errori e in modo da lanciare le opportune eccezioni. I vari predicati lanciano

pertanto un'istanza di `CsharpException` corrispondente all'eventuale errore: tale istanza è catturata nello stato *GoalEvaluation* del motore, mentre le informazioni sull'errore che si è verificato sono poi utilizzate dallo stato *Exception* per ricercare un subgoal `csharp_catch/3` opportuno che sia in grado di gestirlo.

La `CsharpLibrary` presenta predicati implementati sia in C# che in Prolog. Naturalmente il controllo delle condizioni di errore deve avvenire in entrambi i casi e si è adottato lo stesso meccanismo utilizzato in precedenza per le altre librerie: nel primo caso le eccezioni (cioè le opportune istanze di `CsharpException`) vengono lanciate direttamente dai corrispondenti metodi C# ogni qualvolta si verifica un errore, mentre nel secondo caso sono lanciate da metodi "guardia" (sempre espressi in C#) invocati per controllare i parametri prima dell'esecuzione del predicato Prolog.

Le eccezioni possono quindi derivare:

- dall'invocazione di un predicato della `CsharpLibrary` con parametri errati;
- dalla chiamata di un metodo di qualsiasi classe C# esterna al motore.

Naturalmente un utente può decidere di non gestire le eccezioni lanciate dai predicati della `CsharpLibrary`: in questo caso l'esecuzione fallisce e l'estensione è praticamente nulla.

6.2.1 I predicati: differenze con la JavaLibrary

Nel seguito sono descritte le eccezioni lanciate dai predicati della `CsharpLibrary` ed in quali circostanze ciò avviene. E' importante notare che i predicati possono lanciare innumerevoli tipi di eccezioni, visto che sono utilizzati per invocare classi C# esterne al motore; in questa sede sono naturalmente descritte solo le eccezioni "generiche" che si possono verificare

in tutte le chiamate del predicato (ad esempio quelle riguardanti l'invocazione dei predicati con parametri non corretti).

Molti predicati della `CsharpLibrary`, sebbene con compiti simili, si differenziano da quelli della libreria `JavaLibrary` (presente nella versione Java di tuProlog), e possono quindi lanciare eccezioni diverse. Le principali differenze riguardano i predicati per la creazione di oggetti e i predicati per la gestione di array.

➢ Predicati relativi alla creazione di oggetti

In C#, ad esempio, per recuperare il tipo di un oggetto che non appartiene al namespace System, e che quindi non è definito nell'assembly System.dll, è necessario che l'assembly che contiene tale tipo sia caricato al momento dell'esecuzione; pertanto, è stato introdotto un nuovo predicato, `csharp_object/4` avente questo formato:

❖ **`csharp_object(AssemblyName, ClassName, ArgList, ObjId)`**

`AssemblyName`, rappresenta il nome dell'assembly, con estensione .dll, che deve essere caricato, `ClassName` è un atomo Prolog che rappresenta il nome della classe C#, `ArgList` è la lista di argomenti che vengono passati al costruttore e `ObjId` rappresenta un riferimento all'oggetto che viene creato. Questo predicato può quindi lanciare le seguenti eccezioni:

- `TypeLoadException'(Cause, Message, StackTrace)` se `ClassName` non identifica una classe C# valida;
- `NotSupportedException (Cause, Message, StackTrace)` se il costruttore specificato non è stato trovato;
- `ArgumentException (Cause, Message, StackTrace)` se gli argomenti del costruttore non sono validi;

- `FileNotFoundException (Cause, Message, StackTrace)` se `AssemblyName` non identifica una libreria esistente;
- `TargetInvocationException (Cause, Message, StackTrace)` se gli argomenti di `ArgList` non sono ground;
- `Exception(Cause, Message, StackTrace)` se `ObjId` già riferisce un altro oggetto nel sistema.

L'aggiunta di un metodo per la creazione di oggetti ha comportato anche la realizzazione di un nuovo metodo per la creazione di array:

❖ **csharp_array(string assemblyName, string type, int nargs, Term id)**

Tale metodo viene invocato nel caso in cui si deve creare un array di un tipo che non viene definito nel namespace `System`. Esso quindi può lanciare:

- `FileNotFoundException (Cause, Message, StackTrace)` se `assemblyName` non identifica una libreria esistente.

Il predicato `csharp_class/4` permette, invece, la creazione ed il caricamento di una nuova classe C# da un testo sorgente fornito come argomento, secondo il seguente formato:

❖ **csharp_class(ClassSourceText, FullClassName, AssemblyList, ObjId)**

dove `ClassSourceText` è una stringa rappresentante il codice sorgente da compilare, `FullClassName` è il nome completo della classe, `AssemblyList` è una lista di nomi di assembly, con estensione .dll, che devono essere caricati durante la compilazione dinamica, infine, `ObjId` è un riferimento ad un oggetto di tipo `System.Type` che rappresenta la classe appena creata. Questo predicato può lanciare:

- `IOException (Cause, Message, StackTrace;`

  se `AssemblyList` fa riferimento a delle librerie .dll non esistenti

- `IOException(Cause, Message, StackTrace);`

  se `ClassSourceText` contiene errori

- `Exception(Cause, Message, StackTrace)`

  se `ObjId` già riferisce un altro oggetto nel sistema.

In C#, a differenza di Java vengono definite anche le proprietà, che servono per accedere ai valori dei campi di un oggetto. Il predicato `csharp_call/3` è utilizzato, infatti, per invocare un metodo o una proprietà di un oggetto, avente il seguente formato:

❖ **`csharp_call(ObjId, MethodInfo, ObjIdResult)`**

per cui le eccezioni che può lanciare sono:

- `NotSupportedException(Cause, Message, StackTrace)`

  se si invoca un metodo o una proprietà non valida per l'oggetto o per la classe oppure se gli argomenti del metodo non sono validi.

- `AmbiguosMatchException (Cause, Message, StackTrace)`

  se è stata trovata più di una proprietà con il nome specificato.

❖ **`'<-'(ObjId, MethodInfo)`**

Chiama `csharp_call/3`, quindi lancia le sue stesse eccezioni.

❖ **`returns('<-'(ObjId, MethodInfo), ObjIdResult)`**

Chiama `csharp_call/3`, quindi lancia le sue stesse eccezioni.

❖ **csharp_object_bt(ClassName, ArgList, ObjId)**

Chiama `csharp_object/3`, quindi lancia le sue stesse eccezioni.

➢ Predicati relativi alla gestione degli array

La libreria `CSharpLibrary` consente anche di creare e di gestire array da tuProlog .NET. Java permette di impostare e di leggere i valori contenuti nei campi di un array e di conoscere la lunghezza del vettore, invocando semplicemente i metodi `get`, `set` e `getLength` della classe `java.lang.reflect.Array`, passando come argomenti l'oggetto che rappresenta l'array nel caso di una `getLength`, l'oggetto e l'indice nel caso di una `get`, l'oggetto, l'indice e il valore da impostare nel caso di una `set`. Questi metodi vengono poi invocati attraverso il predicato `java_call_3`. In C# invece, la reflection sugli array non è stata sviluppata, perciò tali funzionalità sono state realizzate introducendo *ex nov*o tre metodi appositi di cui devono essere considerate le eccezioni che possono lanciare.

Il metodo utilizzato per l'impostazione di valori è:

❖ **scsharp_array_set(Term obj_id, Term i, Term what)**

dove `obj_id` rappresenta il riferimento all'array, `i` rappresenta la posizione al suo interno e `what` il riferimento all'oggetto che deve essere impostato. Può lanciare le seguenti eccezioni:

- `NullReferenceException(Cause, Message, StackTrace)`

  se `obj_id` non riferisce alcun oggetto del sistema.

- `ArgumentException(Cause, Message, StackTrace)`

  se `obj_id` non è un valore corretto per l'array e se `i` non è un indice corretto

- `IndexOutOfRangeException(Cause, Message, StackTrace)`

  se `i` non è un indice numerico corretto

Il metodo utilizzato per ottenere i valori è:

❖ **`scsharp_array_get(Term obj_id, Term i, Term what)`**

dove `obj_id` rappresenta il riferimento all'array, `i` rappresenta la posizione al suo interno e `what` il riferimento all'oggetto ricavato dal vettore. Può lanciare le seguenti eccezioni:

- `ArgumentException(Cause, Message, StackTrace)`

  se `obj_id` non riferisce alcun oggetto del sistema oppure se `i` non è un indice corretto.

Il metodo utilizzato per poter ricavare la lunghezza di un array è:

❖ **`scsharp_array_lenght 2(Term obj_id, Term what)`**

dove `obj_id` rappresenta il riferimento all'array, e `what` il riferimento al valore che indica la lunghezza. Può lanciare le seguenti eccezioni:

- `ArgumentException(Cause, Message, StackTrace)`

  se `what` non è un valore corretto

- `NullReferenceException(Cause, Message, StackTrace)`

  se `obj_id` non riferisce alcun oggetto del sistema

Per permettere l'invocazione trasparente di questi metodi sono state inserite nella teoria della libreria `CsharpLibrary` le seguenti clausole:

❖ `csharp_array_set(Array,Index,Object):-`

`scsharp_array_set(Array,Index,Object),!.`

❖ `csharp_array_get(Array,Index,Object):-`

`scsharp_array_get(Array,Index,Object),!.`

- `csharp_array_length(Array,Length):-`

  `scsharp_array_length(Array,Length).`

6.2.2 CliLibrary

La `CliLibrary`[7] è un' evoluzione della `CsharpLibrary` che permette di utilizzare e manipolare componenti ed oggetti scritti in qualsiasi altro linguaggio supportato da .NET direttamente da programmi Prolog. Questa possibilità di tuProlog di interoperare con differenti linguaggi è offerta da entità chiamate convenzioni che, occupandosi della gestione delle caratteristiche di ogni specifico linguaggio, permettono di superare disomogeneità e differenze sintattiche.

La `CliLibrary` deve quindi adottare un comportamento comune a prescindere dai linguaggi utilizzati. A questo scopo sono stati introdotti nuovi predicati che permettono di trattare le convenzioni. Altri predicati della `CsharpLibrary` sono stati invece estesi in modo da utilizzare le specifiche convenzioni e quindi presentare funzionamenti differenti a seconda dei parametri passati.

Analogamente a quanto fatto per la `CsharpLibrary`, i predicati della `CliLibrary` possono essere anch'essi modificati in modo da non fallire in caso di errori e possono lanciare opportune eccezioni (istanze di `CsharpException)`. Allo stesso modo nella teoria della libreria è stata aggiunta la clausola `cli_catch/3` che ha gli stessi compiti e la stessa struttura di `csharp_catch/3`. In conclusione è possibile dotare la `CliLibrary` del meccanismo delle eccezioni adottando lo stesso approccio utilizzato per la `CsharpLibrary`.

[7] Marco Albertin, *Progetto e sviluppo del supporto all'interoperabilità fra l'interprete tuProlog e linguaggi su piattaforma Microsoft.NET,* tesi di laurea in Linguaggi e Modelli Computazionali, Bologna, 2008.

7 Piano del Collaudo

Il piano del collaudo ha lo scopo di verificare la correttezza e le performance del sistema a seguito dell'estensione apportata.

tuProlog .NET è stato sottoposto a tre tipi di test:

- test delle eccezioni
- test di retro compatibilità
- test delle prestazioni.

7.1 Test delle eccezioni

E' stato collaudato il funzionamento delle due coppie di nuovi predicati `throw/1` con `catch/3` e `csharp_throw/1` con `csharp_catch/1` rispettivamente mediante i test Throw/Catch e Csharp_Throw/Catch verificando che fossero rispettati tutti i requisiti che lo standard impone per il comportamento di questi predicati. Infine sono stati collaudati tutti i predicati di libreria.

7.1.1 Throw/Catch Test

Per il collaudo è stato utilizzato lo strumento Unit Test, dal quale risulta che i predicati `throw/1` e `catch/3` rispettano tutti i requisiti di funzionamento specificati dallo standard:

- `Handler` deve essere eseguito con le sostituzioni effettuate nel processo di unificazione tra `Error` e `Catcher`

```
Theory: p(0) :- p(1). p(1) :- throw(error).

Goal: atom_length(err, 3), catch(p(0), E, (atom_length(E,
Length), X is 2+3)), Y is X+5.

yes. Y / 10  E / error  X / 5  Length / 5
```

```
Solution:
',(atom_length(err,3),',(catch(p(0),error,',(atom_length(er
ror,5),is(5,'+'(2,3)))),is(10,'+'(5,5))))
```

❖ Deve essere eseguito il più vicino antenato `catch/3` nell'albero di risoluzione il cui secondo argomento unifica con l'argomento di `throw/1`

```
theory: p(0) :- throw(error). p(1).

Goal: catch(p(1), E, fail), catch(p(0), E, atom_length(E,
Length)).

yes. Length / 5  E / error

Solution:
',(catch(p(1),error,fail),catch(p(0),error,atom_length(error,
5)))
```

❖ L'esecuzione deve fallire se si verifica un errore durante l'esecuzione di un goal e non viene trovato nessun nodo `catch/3` il cui secondo argomento unifica con l'argomento dell'eccezione lanciata

```
theory: p(0) :- throw(error).

Goal: catch(p(0), error(X), true).

no
```

❖ `catch/3` deve fallire anche se `Handler` è falso

```
theory: p(0) :- throw(error).

Goal: catch(p(0), E, E == err).

no
```

❖ Se `Goal` è un goal non deterministico, allora deve essere rieseguito; se però durante la sua esecuzione si verifica un errore, allora tutti i punti di scelta relativi a `Goal` devono essere tagliati e `Handler` deve essere eseguito una sola volta

```
Theory: p(0). p(1) :- throw(error). p(2).

Goal: catch(p(X), E, E == error).

yes. X / 0 Solution: catch(p(0),E,'=='(E,error))

yes.  E  /  error  X  /  1  Solution:
catch(p(1),error,'=='(error,error))
```

- `catch/3` deve fallire se si verifica un'eccezione durante l'esecuzione di `Handler`

```
Theory: p(0) :- throw(error).

Goal: catch(p(0), E, throw(err)).

Solution: no
```

7.1.1 Csharp_Throw/Catch Test

Per il collaudo è stato utilizzato lo strumento Unit Test da cui risulta che i predicati `csharp_throw/1` e `csharp_catch/3` rispettano tutti i requisiti di funzionamento specificati dallo standard:

- Verifica che il gestore sia eseguito con le sostituzioni effettuate durante il processo di unificazione tra l'eccezione e il `catcher`, e che successivamente sia eseguito il `finally`. Ad esempio:

```
Goal:atom_length(err,                                          3),
csharp_catch(csharp_object('util.dll','util.Counter', ['MyCounter'],
c), [('NotSupportedException'(Cause, Message, StackTrace), ((X is
Cause+2, 5 is X+3)))], Y is 2+3), Z is X+5.

yes. Z / 7  X / 2  Cause / 0  StackTrace / []  Y / 5  Message /
'Constructor not found: class util.Counter'
```

- Verifica che sia eseguito il più vicino antenato `csharp_catch/3` nell'albero di risoluzione che esso abbia un `catcher` unificabile con l'argomento di `csharp_throw/1`. Ad esempio:

```
csharp_catch(csharp_object('util.dll','util.Counter', ['MyCounter'],
c), [('NotSupportedException'(Cause, Message, StackTrace), true)],
true),          csharp_catch(csharp_object('util.dll','util.Counter',
```

```
['MyCounter2'], c2), [('NotSupportedException'(C, M, ST), X is C+2)], true).

yes.

X / 2  Message / 'Constructor not found: class util.Counter' C / 0  M / 'Constructor not found: class util.Counter'  ST / []  Cause / 0  StackTrace / []
```

❖ Verifica che l'esecuzione fallisca in presenza di un errore durante l'esecuzione di un goal e non sia trovato nessun nodo `csharp_catch/3` avente un `catcher` unificabile con l'argomento dell'eccezione lanciata. Ad esempio:

```
csharp_catch(csharp_object('util.dll','util.Counter', ['MyCounter'], c), [('Exception'(Cause, Message, StackTrace), true)], true).

No.
```

❖ Verifica che `catch/3` fallisca se il gestore è falso. Ad esempio:

```
csharp_catch(csharp_object('util.dll','util.Counter', ['MyCounter'], c), [('NotSupportedException'(Cause, Message, StackTrace), false)], true).";

no.
```

❖ Verifica che il `finally` venga eseguito in caso di successo di `CsharpGoal`. Ad esempio:

```
csharp_catch(csharp_object('System.Collections.ArrayList', [], l), [(E, true)], (X is 2+3, Y is 3+5)).

yes. Y / 8  X / 5
```

❖ Verifica che `catch/3` fallisca al verificarsi di un'eccezione durante l'esecuzione del gestore. Ad esempio:

```
csharp_catch(csharp_object('util.dll','util.Counter', ['MyCounter'], c), [('NotSupportedException'(Cause, Message, StackTrace),
```

```
csharp_object('util.dll''util.Counter',    ['MyCounter2'],    c2))],
true).

no
```

❖ Verifica della correttezza della ricerca del `catcher` all'interno della lista. Ad esempio:

```
csharp_catch(csharp_object('util.dll','util.Counter', ['MyCounter'],
c),   [('Exception'(Cause,   Message,   StackTrace),   X   is   2+3),
('NotSupportedException'(Cause, Message, StackTrace), Y is 3+5)],
true).

yes. Cause / 0  StackTrace / []  Y / 8  Message / 'Constructor not
found: class util.Counter'
```

7.1.2 Test dei predicati di libreria

Il lancio di tutte le possibili eccezioni che i predicati di libreria possono ora generare è stato collaudato provando ad invocare un predicato con dei parametri errati e verificando che il motore lanciasse l'eccezione corretta. Le classi Unit Test che racchiudono i test effettuati nei loro metodi (contrassegnati con l'attributo [TestMethod]) sono:

❖ CsharpLibrary_ExceptionTest: 27test
❖ BasicLibrary_ExceptionTest: 85 test
❖ BuiltIn_ExceptionTest: 49 test
❖ IOLibrary_ExceptionTest: 27 test
❖ ISOLibrary_ExceptionTest: 7 test
❖ DCGLibrary_ExceptionTest: 2 test

Il collaudo ha avuto esito positivo per tutti i 197 test eseguiti.

7.2 Test di retro compatibilità

Durante il collaudo dell'estensione, è stato verificato che il funzionamento del motore non fosse cambiato nei confronti dei test già esistenti per la precedente

versione. Come previsto, la verifica ha avuto un esisto positivo, vista la malleabilità architetturale di tuProlog che ha permesso di estendere il motore attraverso la creazione di un nuovo stato, senza intaccare il funzionamento degli altri stati. Infatti nelle suite di test esistenti non è prevista la generazione e gestione delle eccezioni. Questo test ci garantisce che venga rispettato il requisito fondamentale di correttezza. Quest'ultimo impone che se l'invocazione di un predicato non porta a fallimento prima dell'introduzione del meccanismo delle eccezioni, non deve fallire neanche dopo. Sono stati eseguiti anche dei test sui singoli predicati e si è verificato il loro fallimento se essi venivano invocati con parametri errati.

7.3 Test delle performance

I test delle performance forniscono una stima sulla bontà del sistema. Questi test si dividono in test di velocità di elaborazione, e test di velocità in caso di errore.

7.3.1 Test di velocità di elaborazione

Sono stati effettuati dei test per verificare la velocità di elaborazione dell'applicazione nell'esecuzione di una query. Dopo aver caricato una teoria nel motore viene stabilita una query calcolando il tempo medio (espresso in millisecondi) necessario per risolverla. Questi benchmark sono stati usati anche per altre implementazione di Prolog (SICStus Prolog[8])e sono:

- *Crypt* risolve un puzzle criptoaritmetico;
- *Deriv* differenzia simbolicamente quattro funzioni di una singola variabile;
- *Poly* somma simbolicamente 1+x+y+z alla decima potenza;
- *Primes* cerca tutti i numeri primi fino a 100;

[8]SICStus Prolog Leading Prolog Technology: http://www.sics.se/isl/sicstuswww/site/performance.html

- *Qsort* utilizza l'algoritmo Quicksort per l'ordinamento ricorsivo di una lista di 50 interi usando liste differenti;
- *Queens* cerca tutti i posti sicuri di 9 regine su una scacchiera 9x9;
- *Query* cerca nazioni con densità di popolazione simile;
- Tak è un benchmark artificiale, originariamente scritto in Lisp. Esso è pesantemente ricorsivo e fa molti calcoli semplici con gli interi.

Sono stati effettuati i medesimi test sia sul tuProlog .NET versione 2.1 che nella versione che supporta le eccezioni.

	tuProlog .NET 2.1	tuProlog .NET 3.0
Crypt	76 ms	77 ms
Deriv	3,5 ms	3,5 ms
Nrev	14 ms	14 ms
Poly	894 ms	891 ms
Primes	72 ms	73 ms
QSort	22 ms	22 ms
Queens	108 ms	114 ms
Query	101 ms	102 ms
Tak	4601 ms	4580 ms

Tabella 6 Confronto delle performance tra le due versioni di tuProlog.NET

I risultati di questi test hanno confermato, come ci si aspettava, che le prestazioni tra le due versioni sono del tutto simili in quanto l'introduzione del supporto delle eccezioni è del tutto trasparente nel caso di un comportamento corretto.

7.3.2 Test di velocità in caso di errore

Sono stati effettuati test delle performance in cui si è sottoposto il motore tuProlog alla risoluzione di predicati invocati con parametri errati. Questi stessi predicati sono stati invocati prima singolarmente e poi con la protezione

del predicato `catch/3` in modo da quantificare il ritardo dovuto al processo di gestione dell'errore.

Nel caso di eccezioni `PrologError` è stato confrontato, ad esempio il predicato:

```
arg(X, p(1), 1).
```

con

```
catch(arg(X, p(1), 1), error(instantiation_error, instantiation_error(Goal, ArgNo)), true).
```

Oppure nel caso di eccezioni `CsharpException` è stato confrontato, ad esempio, il predicato:

```
csharp_object('Nonesiste.dll','Qualsiasi', ['MyCounter'], c).
```

con

```
csharp_catch(csharp_object('Nonesiste.dll','Counter', ['MyCounter'], c), [('FileNotFoundException'(Cause, Message, StackTrace), true)], true).
```

Nei primi casi infatti il predicato fallisce e il motore transita semplicemente nello stato *Halt*. Nel caso dei predicati protetti dal costrutto `catch/3`, come già ampiamente discusso nei capitoli precedenti, l'errore viene catturato e gestito nello stato *Exception*. In quest'ultimo caso si è evidenziato un peggioramento delle prestazioni di circa il 15%, tuttavia la velocità di elaborazione può anche peggiorare nel caso in cui la visita all'indietro nell'albero di risoluzione alla ricerca di un subgoal `catch/3`, il cui secondo argomento unifichi con l'argomento dell'eccezione lanciata, sia molto profonda.

8 Conclusioni

Lo scopo della presente tesi è stato quello di reingegnerizzare l'interprete tuProlog al fine di supportare il meccanismo linguistico che Prolog offre per la gestione delle eccezioni.

Per raggiungere questo obiettivo è stato necessario studiare a fondo l'architettura del motore, in modo da implementare questo meccanismo tenendo conto delle caratteristiche di tuProlog, e rispettando le specifiche dello standard ISO/IEC 13211-1. Grazie alla malleabilità architetturale di tuProlog ed alle analogie tra Java e C#, è stato possibile seguire il medesimo approccio adottato nella versione Java di tuProlog. A tal proposito, il lavoro di conversione tra i due linguaggi svolto nel 2007 si è rivelato utile e ben fatto, in quanto è stato possibile proseguire nell'estensione della versione .NET applicando le stesse soluzioni e gli stessi "artifici" utilizzati per colmare le differenze tra Java e C#.

Il supporto alle eccezioni implementato risulta compatibile sia con lo standard, che con le altre implementazioni di riferimento. Oltre all'introduzione delle eccezioni nei predicati Prolog di libreria, le eccezioni sono supportate anche durante l'utilizzo e la manipolazione di oggetti e classi C# all'interno di programmi Prolog: le eccezioni lanciate dagli oggetti C#, generate durante la computazione, non portano più al fallimento dei predicati della `CsharpLibrary` come in precedenza, ma vengono catturate e gestite. Questo approccio è inoltre estendibile anche alla `CliLibrary`, consentendo in tal modo la gestione delle eccezioni pur utilizzando oggetti e componenti scritti in qualsiasi altro linguaggio supportato da .NET.

La gestione delle eccezioni è fornita come optional, in quanto un utente può decidere di non gestirle; in questo caso il motore si comporta come se le eccezioni non ci fossero ritornando quindi al semplice fallimento del

predicato. Questa scelta progettuale offre non solo un servizio aggiuntivo all'utente, ma permette soprattutto la retro compatibilità con i programmi scritti in precedenza che, non gestendo le eccezioni, possono continuare a funzionare. Questo risultato è confermato dai test precedenti che hanno avuto esito positivo anche con il nuovo motore. Il collaudo dell'estensione è stato svolto seguendo le specifiche dello standard e a questo scopo è stato verificato il corretto funzionamento dei predicati `throw/1` e `catch/3` unitamente alla generazione di eccezioni da parte dei predicati di libreria sollecitati con parametri errati. La totalità dei test effettuati ha dato esito positivo.

Bibliografia

[1] *tuProlog Guide*. April 2007. http://www.alice.unibo.it/xwiki/bin/view/Tuprolog/.

[2] Giulio Piancastelli, Alex Benini, Andrea Omicini, Alessandro Ricci. *The Architecture and Design of a Malleable Object-Oriented Prolog Engine*. 23rd ACM Symposium on Applied Computing (SAC2008). 16-20 March 2008.

[3] J.P.E. Hodgson. *Prolog: The ISO Standard Documents*. June 1999. http://pauillac.inria.fr/~deransar/prolog/docs.html.

[4] Christian Nagel, Bill Evjen, *C# 2008 - Guida per lo sviluppatore,* 2008 Milano, Hoepli.

[5] Matteo Iuliani, *Estensione di un interprete Prolog per la gestione delle eccezioni*, 2009 tesi di laurea in Linguaggi e modelli computazionali, Università di Bologna, a.a 2008/2009.

[6] Mattia Generali, *Reingegnerizzazione del supporto per l'interprete tuProlog su piattaforma Microsoft .NET,* tesi di laurea in Linguaggi e modelli computazionali, Università di Bologna, a.a 2007/2008.

[7] Marco Albertin, *Progetto e sviluppo del supporto all'interoperabilità fra l'interprete tuProlog e linguaggi su piattaforma Microsoft.NET,* tesi di laurea in Linguaggi e Modelli Computazionali, Bologna, a.a. 2007/2008

[8] SICStus Prolog Leading Prolog Technology: http://www.sics.se/isl/sicstuswww/site/performance.html