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

Estensione di un interprete Prolog per la gestione delle eccezioni

CANDIDATO:

Matteo Iuliani

RELATORE:

Chiar.mo Prof. Enrico Denti

Anno Accademico 2008/09

Sessione II

Ringraziamenti

Desidero innanzitutto ringraziare il Professor Enrico Denti per i preziosi insegnamenti ricevuti nei suoi corsi e per le numerose ore dedicate alla mia tesi. Inoltre, ringrazio sentitamente Alex Benini e Ivar Orstavik per la loro disponibilità a dirimere i miei dubbi durante questo lavoro.

Inoltre, vorrei esprimere la mia sincera gratitudine a tutti i compagni di corso, in particolare Tommy, Paolo, Luca, Diana, Caccia, Alessio, Andrea e Davide per il loro prezioso apporto durante i vari lavori di gruppo svolti nel corso della mia carriera universitaria, oltre a Primiano per le infinite ore passate a studiare con me in biblioteca.

Un ringraziamento particolare va anche ai miei coinquilini di Via Valle di Preda 7, che in questi anni mi hanno permesso di studiare con serenità, e a tutti gli amici di Oratino che sono venuti a Bologna festeggiare per la mia laurea.

Ringrazio infine con affetto i miei genitori per il sostegno ed il grande aiuto che mi hanno dato sin dall'inizio della mia carriera universitaria, e anche mia sorella, alla quale auguro un giorno di poter scrivere anche lei queste parole.

Indice

Introduzione

L'obiettivo di questa tesi è estendere l'interprete tuProlog al fine di supportare il meccanismo linguistico che Prolog offre per la gestione delle eccezioni.

tuProlog è un motore Prolog basato su Java per le applicazioni e le infrastrutture di Internet. È facilmente distribuibile, minimale, estremamente configurabile, integrato con Java e supporta molti meccanismi di interazione; tuttavia la sua implementazione 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, mentre andrebbe supportato il meccanismo per la gestione delle eccezioni specificato nello standard ISO/IEC 13211-1 e implementato in molti altri interpreti Prolog.

Lo standard prevede due predicati per fare ciò: `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. In pratica il primo predicato ha una semantica simile a quella del `throw` di Java, mentre il secondo è assimilabile a un blocco `try/catch`. Inoltre lo standard 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 implementato come una macchina a stati finiti facilmente estendibile, in cui ogni stato rappresenta un particolare momento dell'esecuzione. In particolare la valutazione di un predicato primitivo avviene nello stato *GoalEvaluation*, ed è in questo stato che si possono verificare eventuali errori. Il progetto dell'estensione dovrà dunque tenere conto di tale organizzazione, e si può facilmente pensare alla realizzazione di un nuovo stato *Exception* in cui si transita in caso di eccezioni e che abbia lo scopo di gestirle. Tutti i predicati delle varie

librerie di tuProlog dovranno quindi essere rivisti, in modo da lanciare le opportune eccezioni in caso di errori invece che fallire semplicemente.

Infine l'ultimo problema da affrontare è la generazione e la gestione di eccezioni Java all'interno di programmi Prolog che utilizzino oggetti Java durante la computazione. Infatti in tuProlog è possibile accedere a qualsiasi oggetto, classe o package Java da un programma Prolog attraverso l'utilizzo dei predicati della `JavaLibrary`. I metodi Java invocati possono naturalmente lanciare eccezioni, e anche in questo caso si deve progettare un meccanismo che renda possibile gestirle.

La tesi è quindi strutturata nel modo seguente:

- il primo capitolo descrive le caratteristiche e l'architettura di tuProlog, soffermandosi in particolar modo sul funzionamento della macchina a stati finiti che è alla base del motore e sottolineando la facilità con cui è possibile modificale tale architettura software
- il secondo capitolo analizza i requisiti di comportamento dei predicati `throw/1` e `catch/3` e descrive la classificazione degli errori in Prolog presente nello standard
- il terzo capitolo analizza lo stato dell'arte, discutendo il supporto alla gestione delle eccezioni presente in tre famosi motori Prolog e in un motore derivato da tuProlog (jTrolog)
- nel quarto capitolo si fa il punto della situazione e si iniziano a delineare le strategie per la soluzione del problema obiettivo della tesi
- il quinto capitolo descrive le modifiche effettuate al motore per renderlo capace di lanciare e gestire le eccezioni
- il sesto capitolo consiste nella parte manualistica della tesi e descrive per ogni predicato delle librerie di tuProlog quali eccezioni lancia è in quali circostanze ciò avviene

- il settimo capitolo affronta infine il problema della generazione e della gestione di eccezioni Java all'interno di programmi Prolog che utilizzino oggetti Java durante la computazione

1 tuProlog

1.1 Caratteristiche

tuProlog [1] è un motore Prolog basato su Java per le applicazioni e le infrastrutture di Internet. Le sue caratteristiche sono:

- è facilmente *distribuibile*: l'esecuzione di tuProlog richiede soltanto la presenza di una macchina virtuale Java e l'invocazione di un singolo file JAR.

- è *minimale*: il motore di tuProlog è un piccolo oggetto Java che implementa soltanto le caratteristiche essenziali di un motore Prolog. Le funzionalità aggiuntive necessarie per l'applicazione possono essere aggiunte al motore all'occorrenza, attraverso il meccanismo delle librerie. è

- è *configurabile*: un semplice ma potente meccanismo permette di caricare nel motore, sia staticamente che dinamicamente, librerie di predicati, funtori e operatori. Le *librerie* estendono le funzionalità del motore di base, e possono sia appartenere alla distribuzione standard di tuProlog, sia possono essere definite *ad hoc* da un utente o da uno sviluppatore. Le librerie pertanto possono essere usate per configurare all'avvio un motore tuProlog, oppure in qualsiasi momento della sua esecuzione.

- è *integrato con Java*: lato Prolog, grazie alla JavaLibrary, ogni entità Java (oggetto, classe, package) può essere rappresentata come un termine Prolog, e utilizzata. Ciò permette ad esempio di utilizzare direttamente nei programmi Prolog package come Swing o JDBC, aumentando notevolmente le funzionalità del motore. Lato Java, invece, un motore tuProlog può essere invocato e usato come un semplice oggetto Java; diversi motori possono essere utilizzati contemporaneamente da un programma Java, ognuno con la sua base di conoscenza e con il suo set di librerie.

- è *interoperabile*: tuProlog supporta l'interazione attraverso TCP/IP e RMI, e può anche essere fornito come un servizio CORBA. Supporta inoltre, in diverse forme, la coordinazione basata su tuple.

1.2 Architettura

Il cuore dell'architettura [2] di tuProlog è una macchina virtuale Prolog minimale, disponibile sotto forma di un oggetto Java 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 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 già citato meccanismo delle librerie.

1.2.1 Gestione delle librerie

Il gestore delle librerie è il componente architetturale responsabile della configurabilità di tuProlog: ha infatti il compito di caricare nel motore, anche mentre questo esegue, librerie di predicati e funtori. Per scelta progettuale tuProlog è un motore minimale, puramente inferenziale; i soli predicati *built-in* definiti sono quelli che influiscono direttamente sul processo di risoluzione (come `cut/0`), quelli troppo basilari per poter essere definiti altrove (come `fail/0`) e quelli che per motivi di efficienza necessitano di essere definiti vicino al "core" (come `','/2`). Questi predicati sono sempre disponibili nel motore, e non possono mai essere scaricati. Qualsiasi altro predicato che si voglia utilizzare deve essere definito in qualche libreria, e questa deve essere stata caricata nel motore dal gestore delle librerie.

Per conferire al motore le classiche funzionalità supportate da qualsiasi altro motore Prolog, la distribuzione standard rende disponibili quattro librerie che sono caricate di default al momento della creazione del motore:

i. `BasicLibrary`, che definisce alcuni predicati e funtori di base che si trovano di solito nei sistemi Prolog, con l'eccezione dei predicati di I/O

ii. `IOLibrary`, che fornisce alcuni dei predicati Prolog standard di I/O: sono separati dagli altri predicati per marcare l'ortogonalità tra i due aspetti della computazione e dell'interazione

iii. `ISOLibrary`, che definisce i predicati standard ISO non definiti nelle precedenti librerie e non appartenenti alla categoria dell'I/O

iv. `JavaLibrary`, che definisce tutti i predicati per l'interazione tra il mondo Java e il mondo Prolog

Le librerie possono essere espresse solo in Java, solo in Prolog, o in entrambi i linguaggi. Nel primo caso la libreria ha un comportamento deterministico, dovuto proprio al linguaggio di implementazione scelto; per introdurre il non-determinismo, e quindi predicati e funtori che possono restituire soluzioni multiple, va aggiunto uno strato Prolog (non deterministico) sopra quello Java (deterministico), con i predicati Prolog che all'occorrenza possono fare uso di quelli Java, ma non viceversa.

1.2.2 Gestione dei predicati primitivi

Nel gergo tuProlog, i predicati Java sono chiamati *predicati primitivi*, quelli Prolog *predicati di libreria*, mentre quelli scritti da un utente nel suo programma Prolog sono chiamati *predicati definiti dall'utente*. Solo i predicati di libreria e quelli definiti dall'utente sono gestiti dal motore inferenziale e dal suo algoritmo di esecuzione, mentre i predicati primitivi sono invocati come metodi Java quando un subgoal ne referenzia uno.

Il componente relativo alla gestione dei predicati primitivi si occupa quindi di distinguere tali predicati dagli altri, e di invocarli correttamente al momento del bisogno attraverso il meccanismo della riflessione.

1.2.3 Gestione delle teorie

Il componente relativo alla gestione delle teorie si occupa di creare la base di conoscenza del motore e di riconoscere le direttive da eseguire immediatamente, a partire dal programma Prolog in formato testuale. Avvalendosi anche di un parser che riconosce le frasi corrette espresse in Prolog, il componente crea gli oggetti corrispondenti ai termini identificati.

Le clausole sono memorizzate in modo da renderne efficiente il reperimento da parte del motore durante l'esecuzione di una query Prolog; inoltre vengono memorizzate informazioni aggiuntive che indicano ad esempio se la clausola è dinamica (cioè può se essere cancellata dalla base di conoscenza) oppure eventualmente il nome della libreria da cui la clausola proviene.

1.2.4 Gestione del motore inferenziale

Infine il componente relativo alla gestione del motore inferenziale ha lo scopo di disaccoppiare la facciata del sistema dalla macchina che realizza il motore inferenziale. L'utente ha infatti una visione semplificata dell'intero sistema, e quando richiede l'esecuzione di una query Prolog il componente si occupa di inoltrare tale query alla macchina a stati finiti che realizza il motore inferenziale. Oltre a questo, il componente definisce alcune funzioni di utilità per la macchina a stati finiti sottostante.

1.3 Il motore inferenziale

Il motore inferenziale di tuProlog è implementato come una macchina a stati finiti, il cui funzionamento è mostrato in Figura 1. La macchina è composta da:

i. uno stato iniziale, che rappresenta l'inizio della procedura di risoluzione di una query

ii. quattro stati intermedi, che rappresentano le attività eseguite durante il processo di risoluzione

iii. quattro stati finali, che rappresentano i differenti risultati dell'esecuzione di una dimostrazione

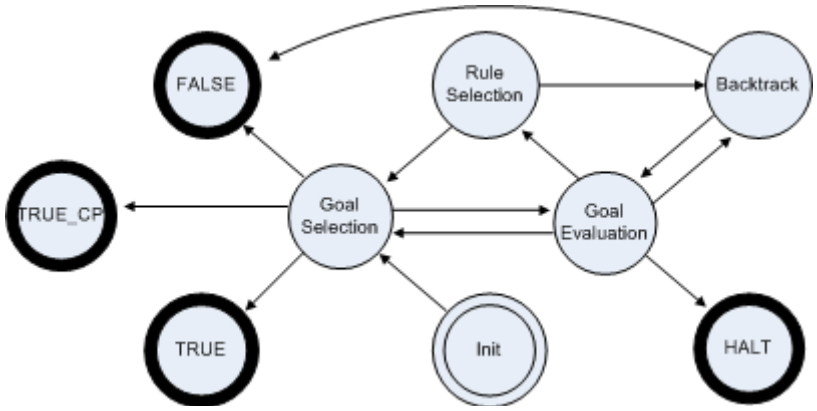

Figura 1: La macchina a stati finiti che implementa il motore inferenziale di tuProlog [2]

1.3.1 Funzionamento

Init è lo stato iniziale della macchina a stati finiti quando questa inizia una dimostrazione: inizializza il motore tuProlog estraendo dalla query i subgoal da valutare e costruisce un oggetto che rappresenta il *contesto di esecuzione* del subgoal attuale; la macchina passa poi allo stato successivo.

Lo stato *GoalSelection* recupera il prossimo goal da valutare dalla lista dei subgoal. Se non ci sono subgoal viene verificata l'esistenza di 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. Se invece è possibile estrarre un subgoal dalla lista, bisogna verificare se può essere valutato: se è un numero la dimostrazione fallisce e il motore si porta nello stato *FALSE*, mentre se è una variabile vengono eseguite una serie di operazioni particolari per permettere alla dimostrazione di procedere correttamente. La macchina si porta 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. Se siamo in presenza di un predicato primitivo, la macchina si porta nello stato *GoalSelection* o *Backtrack*, in base al risultato (successo o fallimento) della valutazione del predicato. Se invece il funtore principale del subgoal non rappresenta un predicato primitivo, la computazione può proseguire soltanto cercando

nella base di conoscenza del motore una clausola compatibile con tale subgoal. 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 nella base di conoscenza del motore delle regole compatibili con il subgoal corrente. Se non ce ne sono, la macchina si porta nello stato *Backtrack* ed inizia il backtracking. Viceversa invece, trovata una regola compatibile, viene preparata la sua valutazione: viene creato un nuovo *contesto di esecuzione*, il subgoal viene unificato con la testa della clausola, mentre il corpo di questa viene aggiunto alla lista dei risolventi sostituendo il subgoal appena unificato. Se l'insieme di regole compatibili ha alternative aperte e non proveniamo da un backtracking, viene creato anche un nuovo contesto per questo punto di scelta, mentre se non ci sono alternative aperte e proveniamo 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* viene subito effettuato un controllo sull'insieme delle clausole compatibili con il subgoal corrispondente all'ultimo punto di scelta aperto; se l'insieme è vuoto la macchina si porta 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* per far ripartire il processo di risoluzione.

1.3.2 Realizzazione

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).

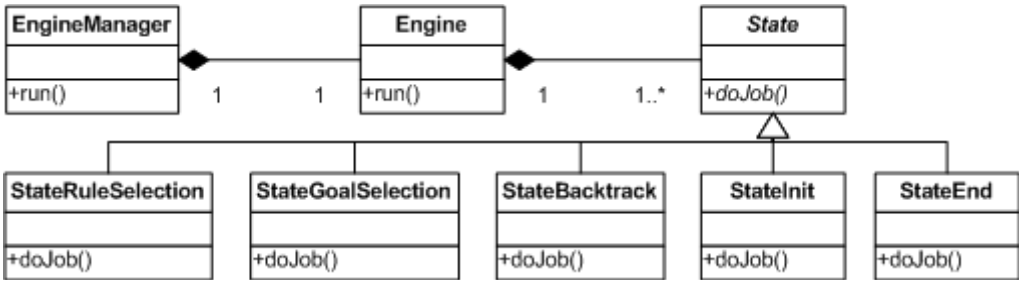

Figura 2: Progetto della macchina a stati finiti che implementa il motore inferenziale di tuProlog attraverso l'utilizzo del pattern State

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. Entrando più nel dettaglio, il pattern è implementato memorizzando nella classe `Engine` il prossimo stato in cui spostarsi, mentre le transizioni tra i diversi stati sono gestite direttamente da questi.

1.4 Malleabilità architetturale

L'architettura di tuProlog appena descritta presenta la caratteristica della *malleabilità*, che fa riferimento alla facilità con cui un'architettura software può essere modificata. La malleabilità non è una proprietà atomica, ma può essere scomposta in tre proprietà (non funzionali):

- *possibilità di evoluzione*: rappresenta il grado con cui un componente architetturale può essere modificato senza influire negativamente sugli altri componenti
- *estensibilità*: è la facilità con cui si possono aggiungere funzionalità al sistema, ed è favorita dal disaccoppiamento tra i componenti architetturali

- *personalizzazione*: rappresenta la possibilità di specializzare il comportamento di un componente architetturale, in modo che un componente personalizzato possa essere utilizzato da un cliente dei suoi servizi specializzati, senza avere ripercussioni negative sugli altri clienti del componente

La malleabilità architetturale di tuProlog deriva quindi dalla suddivisione delle responsabilità tra i diversi componenti gestori, ognuno con le sue responsabilità, e dalla realizzazione del motore inferenziale come macchina a stati finiti. La possibilità di evoluzione consiste nel poter personalizzare il comportamento della macchina al livello di granularità del singolo stato, lasciando intatti gli altri stati. L'estensibilità invece è aiutata dal fatto di poter aggiungere facilmente (grazie all'utilizzo del pattern State) nuovi stati alla macchina, a fronte di nuovi requisiti di funzionamento. Proprio questa caratteristica tornerà molto utile per poter estendere il motore tuProlog in modo da gestire le eccezioni.

2 Eccezioni in ISO Prolog

2.1 Prolog: lo standard ISO

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. Queste incoerenze erano dovute soprattutto al fatto che alcune parti del linguaggio erano ambigue; lo standard risolve questo problema chiarificando ad esempio il significato di unificazione, oppure introducendo una semantica formale per i predicati di modifica della base di conoscenza e per altri predicati.

Oltre a chiarire questi aspetti, lo standard introduce anche:

- un nuovo sistema di predicati per l'I/O che si discosta radicalmente da quelli dell'Edinburgh Prolog (il dialetto alla base della sintassi di molte moderne implementazioni Prolog)
- i costrutti `catch/3` e `throw/1` per la gestione delle eccezioni

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

2.2 Errori e eccezioni in ISO Prolog

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.

Un meccanismo di gestione degli errori dovrebbe essere in grado di rilevare le situazioni di errore e, invece che terminare il programma come diretta conseguenza dell'errore, dovrebbe in modo controllato trasferire 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, cioè all'esterno deve essere o invisibile oppure riportato in maniera piacevole. In Prolog ciò si ottiene per mezzo del predicato `catch/3`
- il principio del salto atomico, che stabilisce che il meccanismo di gestione in caso di errore deve essere in grado di uscire in una sola operazione da un numero arbitrario contesti di esecuzione annidati. In Prolog ciò si ottiene per mezzo del predicato `throw/1`

Quindi il predicato `throw(Error)` serve per lanciare un'eccezione, mentre il predicato `catch(Goal, Catcher, Handler)` può essere utilizzato per prevenire la terminazione forzata di un programma Prolog causata da eventuali errori.

Il sistema mette in esecuzione `Goal`, e nel caso si verifichi un errore durante l'esecuzione:

i. il subgoal in cui si è verificato l'errore viene sostituito dal subgoal `throw/1`

ii. viene ricercato 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`

iii. quando questa clausola viene trovata, il percorso nell'albero di risoluzione che ha dato origine all'eccezione viene tagliato

iv. il catcher viene rimosso, perché è valido soltanto per il goal protetto e non per il gestore

v. il gestore viene eseguito

2.3 I predicati ISO `throw/1` e `catch/3`

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

Il predicato `catch(Goal, Catcher, Handler)` serve invece per proteggere l'esecuzione di `Goal` da eventuali eccezioni. Il predicato si comporta come `call(Goal)` se non vengono lanciate eccezioni durante l'esecuzione di `Goal`. Se invece durante tale esecuzione viene lanciata un'eccezione attraverso `throw/1`, il sistema provvede a tagliare tutti i punti di scelta generati da `Goal` e prova ad unificare `Catcher` con l'argomento di `throw/1`. Se l'unificazione ha successo, esegue `call(Handler)` mantenendo le sostituzioni effettuate nell'unificazione. Se invece fallisce, il sistema continuerà a risalire l'albero di risoluzione in cerca di una clausola `catch/3` che unifichi. Se tale clausola non esiste, il predicato fallisce. L'esecuzione riprende infine con il subgoal successivo a `catch/3`. In caso di eccezioni gli effetti collaterali che si sono eventualmente verificati durante l'esecuzione di `Goal` (come ad esempio le modifiche alla base di conoscenza del sistema) non vengono comunque distrutti, ugualmente a quanto accade in caso di fallimento di un predicato.

Pertanto `catch/3` è vero se:

i. `call(Goal)` è vero, oppure

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

Bisogna poi precisare altri due aspetti:

- se `Goal` è un predicato non-deterministico, allora attraverso il backtracking può capitare che venga rieseguito; tuttavia `Handler`

viene eventualmente eseguito una sola volta, perché 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 l'esecuzione di `Handler`: il gestore quindi non è protetto

Chiariamo infine con un esempio il funzionamento dei due predicati `throw/1` e `catch/3`. Dato il programma:

```
p(X):- throw(error), write('---').

p(X):- write('+++').
```

l'esecuzione di:

```
catch(p(0), E, write(E)), fail.
```

produce l'output:

```
error
```

e non:

```
error---
```

perché, una volta lanciata l'eccezione, viene abortita l'esecuzione di `p(X)` e dopo l'esecuzione del gestore dell'eccezione si prosegue con `fail/0`, che è il subgoal successivo a `catch/3`.

Inoltre non viene neanche stampato:

```
+++
```

perché l'unificazione con successo di `Catcher` con l'eccezione taglia tutti i punti di scelta relativi a `p(X)`.

2.4 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, mentre `Implementation_defined_term` è un termine non standard caratteristico di ogni implementazione, e che può anche essere omesso. 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 nel seguito vengono analizzati i differenti tipi di errore che si possono presentare.

2.4.1 Errori di istanziazione

`instantiation_error`

Si ha un errore di istanziazione quando un argomento di un predicato o uno dei suoi componenti è una variabile, mentre è necessario che sia istanziato.

Ad esempio, il goal:

```
X is Y+1
```

in caso `Y` non sia istanziato presenta un errore di istanziazione perché il secondo argomento di `is/2` contiene una variabile.

2.4.2 Errori di tipo

`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, ma non è neanche una variabile (in qual caso si dovrebbe ha un errore di istanziazione).

`ValidType` è il tipo di dato previsto, `Culprit` è il valore (errato) trovato.

`ValidType` può essere uno dei seguenti atomi: `atom`, `atomic`, `byte`, `callable`, `character`, `evaluable`, `in_byte`, `in_character`, `integer`, `list`, `number`, `predicate_indicator`, `variable`.

Ad esempio, supponiamo di avere a disposizione un predicato

```
date_plus(NumberOfDays, Date1, Date2)
```

che è vero se `Date1` e `Date2` sono due date della forma `date(Year, Month, Day)` e `NumberOfDays` è il numero di giorni compresi tra `Date1` e `Date2`. Il goal:

```
date_plus(27, date(2009, march, 11), X)
```

presenta l'errore di tipo `type_error(integer, march)` perché il mese deve essere espresso come un intero.

2.4.3 Errori di dominio

```
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). Ad esempio, se utilizziamo degli interi da 1 a 12 per rappresentare i mesi dell'anno e in un predicato riferiamo un mese con l'intero 13, siamo in presenza di un errore di dominio.

`ValidDomain` è il dominio previsto, `Culprit` è il valore (errato) trovato.

`ValidDomain` può essere uno dei seguenti atomi: `character_code_list`, `close_option`, `flag_value`, `io_mode`, `not_empty_list`, `not_less_than_zero`, `operator_priority`, `operator_specifier`, `prolog_flag`, `read_option`, `source_sink`, `stream`, `stream_option`, `stream_or_alias`, `stream_position`, `stream_property`, `write_option`.

2.4.4 Errori di esistenza

```
existence_error(ObjectType, Culprit)
```

Si ha un errore di esistenza quando un predicato cerca di accedere a un oggetto che non esiste, ad esempio un file.

`ObjectType` è il tipo dell'oggetto non esistente, e `Culprit` il suo nome.

`ObjectType` può essere uno dei seguenti atomi: `procedure`, `source_sink`, `stream`.

Ad esempio, se proviamo ad accedere al file `'usr/parent/stella'` che non esiste, l'errore può essere rappresentato come `existence_error(stream, 'usr/parent/stella')`.

2.4.5 Errori di permesso

`permission_error(Operation, ObjectType, Culprit)`

Si ha un errore di permesso quando non si ha il permesso di eseguire l'operazione `Operation` sull'oggetto `Culprit` di tipo `ObjectType`.

`Operation` può essere uno dei seguenti atomi: `access`, `create`, `input`, `modify`, `open`, `output`, `reposition`.

`ObjectType` può essere uno dei seguenti atomi: `binary_stream`, `flag`, `operator`, `past_end_of_stream`, `private_procedure`, `static_procedure`, `source_sink`, `stream`, `text_stream`.

2.4.6 Errori di rappresentazione

`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.

`Flag` può essere uno dei seguenti atomi: `character`, `character_code`, `in_character_code`, `max_arity`, `max_integer`, `min_integer`.

2.4.7 Errori di valutazione

`evaluation_error(Error)`

Si ha un errore di valutazione quando la valutazione di un funtore produce un valore eccezionale.

`Error` può essere uno dei seguenti termini: `float_overflow`, `int_overflow`, `undefined`, `underflow`, `zero_divisor`.

2.4.8 Errori di risorsa

`resource_error(Resource)`

Si ha un errore di risorsa quando il motore Prolog ha insufficienti risorse

per completare l'esecuzione. Ad esempio il programma può aver esaurito la memoria a disposizione, oppure ha raggiunto il limite di file aperti contemporaneamente aperti imposto dal sistema operativo.

`Resource` può essere un termine qualsiasi, ad esempio `memory`.

2.4.9 Errori di sintassi

`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 per qualche altro motivo. Sono errori che si verificano quindi principalmente nei predicati come `read/1`.

`Message` può essere un qualsiasi termine (anche composto o stringa) che descrive l'errore che si è verificato.

2.4.10 Errori di sistema

`system_error`

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

3 Analisi dello stato dell'arte

In questo capitolo verrà discusso il supporto alla gestione delle eccezioni presente in tre famosi motori Prolog (CIAO Prolog, SICStus Prolog e SWI-Prolog) e in un motore derivato da tuProlog (jTrolog).

3.1 CIAO Prolog

CIAO Prolog [4] per la gestione delle eccezioni implementa anche il predicato `intercept/3` (non ISO) oltre ai predicati standard `throw/1` e `catch/3` il cui funzionamento è stato discusso nel capitolo 2.

`intercept(Goal, Catcher, Handler)` prova ad eseguire `Goal`. Se durante tale esecuzione viene lanciata un'eccezione, il sistema prova ad unificare `Catcher` con l'errore, e se l'unificazione ha successo viene eseguito prima `Handler`, poi il predicato successivo a quello che ha generato l'eccezione.

La differenza tra `catch/3` e `intercept/3` sta proprio nel predicato che viene messo in esecuzione dal sistema subito dopo il gestore. Nel predicato standard l'intera esecuzione derivante da `Goal` viene abortita se viene lanciata un'eccezione, e dopo l'esecuzione del gestore dell'eccezione si riprende con il predicato successivo a `catch/3`. Nel caso invece di `intercept/3`, l'esecuzione derivante da `Goal` non viene abortita e dopo il gestore viene eseguito il predicato successivo (derivante da `Goal`) a quello che ha generato l'eccezione.

Ad esempio, dato il programma:

```
p(X):- throw(error), write('---').

p(X):- write('+++').
```

l'esecuzione di:

```
catch(p(0), E, write(E)), fail.
```

produce l'output:

```
error
```

perché, una volta lanciata l'eccezione, viene abortita l'esecuzione di `p(X)` e dopo l'esecuzione del gestore dell'eccezione si prosegue con `fail/0`, che è il predicato successivo a `catch/3`.

Invece l'esecuzione di:

```
intercept(p(0), E, write(E)), fail.
```

produce l'output:

```
error---
```

perché, una volta lanciata l'eccezione, non viene abortita l'esecuzione di `p(X)` e dopo l'esecuzione del gestore dell'eccezione si prosegue con `write('---')`, che è il predicato successivo a quello che ha generato l'eccezione, ovvero `throw/1`.

Naturalmente in nessuno dei due casi viene stampato anche:

```
+++
```

perché l'unificazione con successo di `Catcher` con l'eccezione taglia tutti i punti di scelta relativi a `p(X)`.

3.2 SICStus Prolog

SICStus Prolog [5] per la gestione delle eccezioni prevede anche i predicati `raise_exception/1` e `on_exception/3` (entrambi non ISO) oltre a `throw/1` e `catch/3`. Questi due predicati hanno però esattamente lo stesso comportamento dei predicati standard, non c'è differenza di comportamento come nel caso di CIAO Prolog. La documentazione del sistema è molto completa: descrive infatti per ogni predicato built-in tutti i tipi di eccezioni che può lanciare.

Per quanto riguarda invece la classificazione degli errori, SICStus Prolog introduce due nuove classi di errore oltre alle dieci standard descritte nel paragrafo 2.4: gli *errori di contesto* e gli *errori di consistenza*.

3.2.1 Errori di contesto

`context_error(ContextType, CommandType)`

Si ha un errore di contesto quando l'operazione (goal o dichiarazione) `CommandType` non è permessa nel contesto `ContextType`; in pratica l'operazione è stata chiamata "nel posto sbagliato".

`CommandType` e `ContextType` sono termini che descrivono rispettivamente l'operazione richiesta e il tipo di contesto nel quale l'esecuzione è stata tentata.

3.2.2 Errori di consistenza

`consistency_error(Culprit1, Culprit2, Message)`

Si ha un errore di consistenza quando due valori o operazioni (`Culprit1` e `Culprit2`) sono inconsistenti tra di loro.

`Culprit1` e `Culprit2` sono i due valori/operazioni in conflitto tra di loro, `Message` può essere un qualsiasi termine (anche composto o stringa) che descrive l'errore che si è verificato (se non ci sono informazioni aggiuntive oppure non sono significative può essere `0` oppure `''`).

3.2.3 Il termine `SICStus_Error`

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, mentre `Implementation_defined_term` è un termine non standard caratteristico di ogni implementazione, e che può anche essere omesso. SICStus Prolog sceglie di non omettere tale termine (chiamato `SICStus_Error`), anzi la documentazione descrive la sua struttura esatta per ogni classe di errore. Il termine `SICStus_Error` contiene di solito le informazioni presenti nel corrispondente termine `Error_Term`, più qualche informazione aggiuntiva, come il goal e il numero dell'argomento (la numerazione inizia da 1) che ha causato l'errore. Nel seguito è presentata la struttura del termine `SICStus_Error` per le varie classi di

errore, standard e non standard; i termini sottolineati sono quelli che forniscono informazioni aggiuntive rispetto al corrispondente termine `Error_Term`.

Errori di istanziazione

`instantiation_error(Goal, ArgNo)`

`Goal` è il goal in cui si è verificato l'errore, `ArgNo` è un intero non negativo che indica quale argomento ha causato il problema (`0` indica che il problema non può essere circoscritto a un singolo argomento).

Ad esempio, il goal:

```
X is Y+1
```

in caso `Y` non sia istanziato presenta un errore di istanziazione perché il secondo argomento di `is/2` contiene una variabile. Il corrispondente `SICStus_Error` può essere `instantiation_error(2298 is 2301+1, 2)`.

Errori di tipo

`type_error(Goal, ArgNo, ValidType, Culprit)`

`Goal` è il goal in cui si è verificato l'errore, `ArgNo` è un intero non negativo che indica quale argomento ha causato il problema.

Ad esempio, supponiamo di avere a disposizione un predicato:

```
date_plus(NumberOfDays, Date1, Date2)
```

che è vero se `Date1` e `Date2` sono due date della forma `date(Year, Month, Day)` e `NumberOfDays` è il numero di giorni compresi tra `Date1` e `Date2`. Il goal:

```
date_plus(27, date(2009, march, 11), X)
```

genera il `SICStus_Error` `type_error(date_plus(27, date(2009, march, 11), 235), 2, integer, march)` perché si è verificato un errore nel secondo argomento del goal (il mese deve

essere espresso come un intero, mentre qui è stato espresso attraverso l'atomo `march`).

Errori di dominio

`domain_error(Goal, ArgNo, ValidDomain, Culprit)`

`Goal` è il goal in cui si è verificato l'errore, `ArgNo` è un intero non negativo che indica quale argomento ha causato il problema.

Errori di esistenza

`existence_error(Goal, ArgNo, ObjectType, Culprit, Message)`

`Goal` è il goal in cui si è verificato l'errore, `ArgNo` è un intero non negativo che indica quale argomento ha causato il problema, `Message` può essere un qualsiasi termine (anche composto o stringa) che fornisce informazioni aggiuntive suggerite dal sistema operativo sull'errore che si è verificato (se non ce ne sono oppure non sono significative può essere `0` oppure `''`).

Ad esempio, se proviamo ad accedere al file `'usr/parent/stella'` che non esiste, si il corrispondente `SICStus_Error` può essere `existence_error(see('usr/parent/stella'), 1, stream, 'usr/parent/stella', errno(45))`, dove `errno(45)` è una rappresentazione dell'errore di sistema `ENOENT` (`no such file or directory`).

Errori di permesso

`permission_error(Goal, Operation, ObjectType, Culprit, Message)`

`Goal` è il goal in cui si è verificato l'errore, `Message` può essere un qualsiasi termine (anche composto o stringa) che fornisce informazioni aggiuntive suggerite dal sistema operativo sull'errore che si è verificato (se non ce ne sono oppure non sono significative può essere `0` oppure `''`).

Errori di rappresentazione

`representation_error(Goal, ArgNo, Flag)`

`Goal` è il goal in cui si è verificato l'errore, `ArgNo` è un intero non negativo che indica quale argomento ha causato il problema.

Errori di valutazione

`evaluation_error(Goal, ArgNo, Error)`

`Goal` è il goal in cui si è verificato l'errore, `ArgNo` è un intero non negativo che indica quale argomento ha causato il problema.

Errori di risorsa

`resource_error(Goal, Resource)`

`Goal` è il goal in cui si è verificato l'errore, oppure `0` se nessun goal è specifico responsabile dell'errore (ad esempio se si esaurisce la memoria a disposizione del programma, questo evento potrebbe non essere imputabile ad alcun goal in particolare).

Errori di sintassi

`syntax_error(Goal, Position, Message, Left, Right)`

`Goal` è il goal in cui si è verificato l'errore, `Position` identifica la posizione dello stream da cui è iniziata la lettura, `Left` e `Right` sono rispettivamente le liste di token prima e dopo l'errore.

Errori di sistema

`system_error(Message)`

`Message` può essere un qualsiasi termine (anche composto o stringa) che fornisce informazioni aggiuntive suggerite dal sistema operativo sull'errore che si è verificato (se non ce ne sono oppure non sono significative può essere `0` oppure `''`).

Errori di contesto (non standard)

`context_error(Goal, ContextType, CommandType)`

`Goal` è il goal in cui si è verificato l'errore.

Errori di consistenza (non standard)

`consistency_error(Goal, Culprit1, Culprit2, Message)`

`Goal` è il goal in cui si è verificato l'errore.

3.3 SWI-Prolog

SWI-Prolog [6] invece per la gestione delle eccezioni implementa soltanto i predicati `throw/1` e `catch/3`, esattamente nelle specifiche previste dallo standard ISO. Molti dei predicati built-in in caso di errori generano eccezioni invece che far fallire l'esecuzione, anche se alcuni predicati ancora presentano quest'ultimo tipo di comportamento. Inoltre la documentazione specifica che l'overhead introdotto dalla chiamata di un goal attraverso `catch/3` è pressappoco identico rispetto a quello introdotto dalla chiamata dello stesso attraverso `call/1`; naturalmente in caso di recovery da un eccezione i tempi si allungano, dipendentemente dal gestore definito.

3.4 jTrolog

jTrolog [7] è un motore Prolog derivato da tuProlog. Il sistema di librerie è pressappoco lo stesso, mentre l'implementazione dei termini Prolog e del motore inferenziale di base è sostanzialmente differente. Queste modifiche hanno l'effetto di rendere il motore decisamente più performante (jTrolog risolve l'*Einstein riddle* quasi dieci volte più velocemente di tuProlog), a scapito però della malleabilità architetturale (tuProlog può essere esteso molto più facilmente).

3.4.1 Il motore inferenziale

Il motore inferenziale di jTrolog per risolvere una query necessita di quattro risorse:

- uno stack
- una BindingsTable
- l'accesso ai predicati di libreria primitivi (metodi di libreria)
- l'accesso al database dei predicati ordinari (regole)

Lo stack rappresenta l'albero di ricerca che il sistema costruisce per risolvere la query, ed è costituita da un vettore di ChoicePoint, uno per ogni metodo di libreria o regola nell'albero di ricerca.

La BindingsTable invece tiene traccia dei legami tra le variabili e i termini, e a quale punto dello stack tale legame risale.

Durante il processo di risoluzione, il motore commuta tra sei stati: *Eval*, *Rule*, *Back*, *True*, *True_all*, *False*.

- *True_all* è lo stato finale che viene raggiunto quando il motore ha terminato l'esecuzione con successo
- *False* è lo stato finale che viene raggiunto quando il motore ha terminato l'esecuzione con fallimento
- si entra nello stato *Eval* non appena un nodo ChoicePoint viene aggiunto allo stack. Se il ChoicePoint contiene un predicato che rappresenta un metodo di libreria, questo viene valutato attraverso il meccanismo della riflessione, come in tuProlog. Altrimenti se il ChoicePoint contiene un predicato che rappresenta una regola, il motore passa nello stato *Rule*
- lo stato *Rule* ha la responsabilità di provare a unificare un predicato con una regola presente nella base di conoscenza del motore. Mantiene traccia di quali regole che sono state provate, in modo da non ripeterne l'esecuzione
- quando un predicato primitivo o una regola fallisce, il motore passa nello stato *Back* e inizia il backtracking. Il motore attraversa lo stack ricercando il primo ChoicePoint che ha ancora regole non tentate. Se nello stack non sono presenti alternative aperte, il motore termina l'esecuzione nello stato *False*
- similmente, quando un predicato primitivo o una regola è vera, il motore passa nello stato *True* e attraversa lo stack per assicurarsi di aver risolto tutte le diramazioni dell'albero di risoluzione. Se non ci

sono ulteriori diramazioni da risolvere, il motore termina l'esecuzione nello stato *True_all*

Si può notare in definitiva un sistema di stati simile a quello di tuProlog, anche se con alcune differenze nelle funzionalità dei vari stati. Tuttavia le differenze sostanziali stanno nell'implementazione: mentre in tuProlog si utilizza il pattern State per modellare il comportamento dei diversi stati, in jTrolog le transizioni tra i vari stati e il comportamento del motore in ognuno di questi sono definiti da metodi privati della classe `Engine` che modella il motore (e quindi si hanno i metodi `eval()`, `rule()`, `back()` e `true()`). È questa un'organizzazione molto più semplice del motore inferenziale, ma che d'altro canto presenta minori possibilità di estensione.

Inoltre è previsto anche un metodo privato `cut()` che viene chiamato quando il motore incontra il corrispondente predicato built-in; tale metodo si occupa di rimuovere dallo stack tutte le alternative alla regola corrente.

3.4.2 Gestione delle eccezioni

jTrolog non supporta alcun meccanismo di gestione delle eccezioni, in quanto i progettisti hanno sempre messo in primo piano l'efficienza del motore;: l'introduzione di questo tipo di meccanismo avrebbe infatti portato ad un piccolo rallentamento delle sue prestazioni. Tuttavia, seppur senza implementarla, hanno definito a grandi linee una possibile strategia per lanciare e gestire le eccezioni in jTrolog.

In sostanza si tratta di aggiungere nella classe che implementa il motore inferenziale due nuovi metodi speciali (`try()` e `throw()`), da utilizzare allo stesso modo di come viene gestito il predicato di cut (ad ogni occorrenza di tale predicato il motore chiama il metodo speciale `cut()`). I metodi `try()` e `throw()` dovranno essere chiamati dal motore ad ogni occorrenza rispettivamente dei predicati `catch/3` e `throw/1`.

Il metodo `try()` deve prevedere:

- un blocco *try*, in cui si tenta l'esecuzione del predicato espresso dal primo argomento di `catch/3`

- un blocco *catch*, che serve a catturare e gestire eventuali eccezioni lanciate all'interno del blocco *try*

Il metodo `throw()` deve invece distruggere tutti i legami delle variabili effettuati durante l'esecuzione del goal protetto, quindi lanciare un'eccezione del tipo opportuno.

Tuttavia, anche se in questa sede il funzionamento di jTrolog non è stato analizzato in profondità, la strategia descritta per aggiungere il meccanismo delle eccezioni non è percorribile nel caso di tuProlog, dato che il motore inferenziale di quest'ultimo ha una struttura sostanzialmente diversa e soprattutto più complicata.

4 Eccezioni in tuProlog

4.1 Estensione dell'architettura di tuProlog

In [2] viene proposta una estensione dell'architettura di tuProlog per introdurre nel motore il meccanismo della gestione degli errori. Tale estensione può avvenire attraverso una evoluzione del motore inferenziale descritto nel paragrafo 1.3, introducendo un nuovo stato nella macchina a stati finiti che ne è alla base, come illustrato in Figura 3.

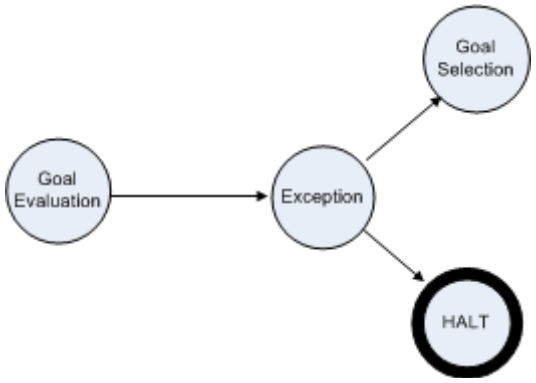

Figura 3: L'estensione della macchina a stati finiti per supportare il meccanismo della gestione degli errori [2]

Il nuovo stato prende il nome di *Exception*, e la macchina vi entra quando nello stato *GoalEvaluation* si verifica un errore durante la valutazione di un subgoal. 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 in testa alla

lista dei subgoal da eseguire, come definito dal terzo argomento di `catch/3`. Il gestore deve inoltre essere preparato per l'esecuzione, mantenendo le sostituzioni effettuate durante il processo di unificazione tra l'argomento di `throw/1` e il secondo argomento di `catch/3`. Infine la macchina transita nello stato *GoalSelection* se è stato trovato un nodo `catch/3` appropriato e quindi l'errore può essere gestito, oppure nello stato *HALT* se tale nodo non è stato trovato: l'errore pertanto non può essere gestito e la macchina termina l'esecuzione.

Per rendere tuProlog capace di gestire gli errori bisogna pertanto:

- modificare la macchina a stati finiti che è alla base del motore
- implementare i due predicati standard `throw/1` e `catch/3`

Naturalmente i vari predicati di libreria devono essere rivisti in modo da generare le opportune eccezioni in caso di errori, invece che fallire semplicemente. Un altro problema che va poi affrontato, legato sempre al supporto alla programmazione multi-paradigma Java/Prolog offerto da tuProlog, è la gestione di eccezioni Java all'interno di programmi Prolog che utilizzano oggetti Java durante la computazione. Quindi implementare in tuProlog il meccanismo della gestione delle eccezioni comprende diversi aspetti, ognuno dei quali verrà discusso in dettaglio nel seguito.

In ogni caso è risultata evidente la malleabilità architetturale di tuProlog, dovuta alla possibilità di evoluzione del suo motore inferenziale senza impattare sugli altri componenti dell'architettura: le modifiche restano infatti confinate all'interno della macchina a stati finiti.

4.2 Considerazioni implementative

Mentre già molto si è detto relativamente a come implementare il nuovo stato *Exception* e i predicati `throw/1` e `catch/3`, in questa sede si ritiene opportuno fare alcune considerazioni relative alla generazione di eccezioni all'interno dei predicati di libreria e alla gestione di eccezioni Java all'interno di programmi Prolog che utilizzano oggetti Java durante la computazione.

4.2.1 Generazione di eccezioni 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) si riescono tipicamente a riscontrare solo durante l'esecuzione. Inoltre ciò deve avvenire sia per i predicati implementati in Java, sia per quelli implementati in Prolog.

Visto che nell'architettura di tuProlog lo strato Prolog è costruito sopra lo strato Java, nel caso dei predicati di libreria implementati in Java il controllo può avvenire soltanto in Java. Nel caso invece dei predicati implementati in Prolog la presenza di errori può essere controllata sia lato Prolog (ad esempio utilizzando meta-predicati per il controllo dei parametri, come `integer/1`), sia lato Java. È chiaro che la seconda alternativa è preferibile per velocizzare la risposta del motore.

Pertanto una soluzione possibile è quella di prevedere delle guardie apposite per i predicati Prolog, ovvero dei metodi Java 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. Nell'implementazione di questa soluzione si dovrebbe comunque trovare il giusto compromesso tra velocità di esecuzione e chiarezza, visto che è necessario spezzare la definizione dei predicati in due parti: il corpo effettivo del predicato espresso in Prolog e la sua guardia espressa in Java.

4.2.2 Eccezioni Java all'interno di programmi Prolog

Volendo effettuare un paragone tra la gestione delle eccezioni in Prolog e in Java, si può notare che in Prolog non è presente il costrutto `finally` che troviamo invece in Java. Consideriamo infatti il seguente codice Java 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 non c'è niente in `catch/3` che corrisponda a `(4)`. Si può ovviare a ciò considerando il blocco `finally` come una serie di operazioni che devono essere eseguite comunque, sia che ci siano eccezioni sia che 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();
}
```

La corrispondenza tra:

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

è sempre valida, mentre `doSomething()` corrisponde a del codice Prolog presente sia in `Goal` che in `Handler`.

Un'altra differenza è che in Java 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 gerarchica. Invece il secondo termine del predicato Prolog `catch/3` 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 ad esempio in un programma Prolog utilizziamo oggetti Java attraverso la `JavaLibrary`.

Per preservare la semantica di Java si può quindi pensare all'implementazione di due predicati aggiuntivi da utilizzare proprio per lanciare e gestire le eccezioni Java in programmi Prolog: `java_throw/1` e `java_catch/3`.

Il predicato `java_throw/1` ha la forma:

`java_throw(`*`java_exception`*`(Cause, Message, StackTrace))`

dove l'atomo *`java_exception`* prende il nome della classe della specifica eccezione Java da lanciare espressa sotto forma di stringa (ad esempio `'java.io.FileNotFoundException'`), mentre i tre parametri rappresentano le tre parti che caratterizzano la tipica eccezione Java.

Invece il predicato `java_catch/3` ha la forma:

```
java_catch(Goal, [(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 Java. Il terzo argomento ha invece la semantica del `finally` di Java e rappresenta il predicato da eseguire alla fine di `Goal` o di uno dei gestori; in caso non fosse necessario si può passare come terzo argomento un termine speciale (es. `0` o `''`). 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 `java_catch/3`.

In definitiva la soluzione ideale è avere due coppie di predicati diversi per trattare gli errori Prolog da un lato (attraverso i predicati `throw/1` e `catch/3`) e le eccezioni Java dall'altro (attraverso `java_throw/1` e `java_catch/3`), con questi ultimi che permettono di avere una semantica più simile a quella del mondo Java e che vanno usati lato Prolog quando si utilizzano oggetti Java attraverso la `JavaLibrary`.

5 Estensione del motore

Per rendere tuProlog capace di gestire gli errori bisogna quindi:

i. modificare la macchina a stati finiti che è alla base del motore

ii. implementare i due predicati standard `throw/1` e `catch/3`

Il primo punto implica la modifica del comportamento dello stato *GoalEvaluation* e la creazione del nuovo stato *Exception*. Come mostrato in Figura 4, questo stato è raggiungibile soltanto a partire dallo stato *GoalEvaluation*; grazie alla malleabilità architetturale con cui è stato progettato tuProlog e al fatto che le transizioni tra i diversi stati sono gestite direttamente da questi, è possibile estendere il motore modificando soltanto i due stati coinvolti nella transizione, lasciando intatti tutti gli altri.

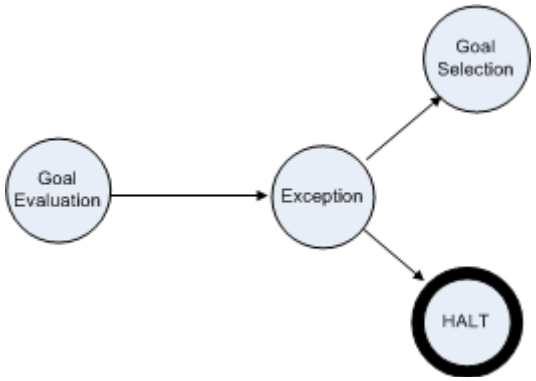

Figura 4: L'estensione della macchina a stati finiti per supportare il meccanismo della gestione degli errori [2]

Per ottenere il secondo punto è necessario invece modificare la classe di tuProlog corrispondente alla libreria in cui si sceglie di inserire i due predicati.

Nel seguito sono analizzate nel dettaglio tutte le modifiche effettuate al motore per renderlo capace di gestire gli errori che si possono verificare durante l'esecuzione.

5.1 La classe `PrologError`

La classe `PrologError` è una classe molto semplice che estende `Throwable` e cattura il concetto di errore Prolog. Una opportuna istanza di `PrologError` viene lanciata ogni volta che un predicato rileva un errore durante la sua esecuzione, e il suo stato consiste nel termine Prolog che rappresenta l'argomento di `throw/1`. Fornisce inoltre una serie di metodi di utilità chiamati dai predicati delle varie librerie per lanciare le corrette istanze di `PrologError` corrispondenti all'errore che si è verificato.

5.2 La classe `EngineManager`

Il motore inferenziale di tuProlog è realizzato attraverso il pattern State (Figura 5).

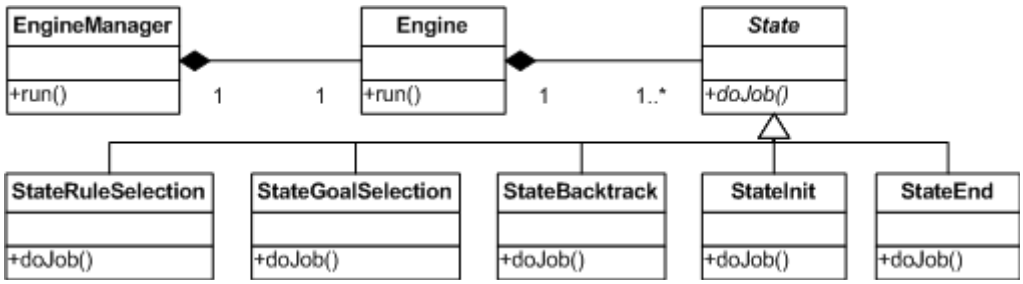

Figura 5: La macchina a stati finiti che implementa il motore inferenziale di tuProlog (prima dell'estensione)

L'entità che cambia stato durante l'esecuzione è il motore inferenziale `Engine`, tuttavia le istanze effettive dei diversi stati sono memorizzati nella classe `EngineManager`, che è la classe che gestisce il motore inferenziale `Engine`. Bisogna quindi aggiungere alla classe `EngineManager` un riferimento a un'istanza della classe che rappresenta il nuovo stato *Exception*.

5.3 La classe `StateGoalEvaluation`

Lo stato *GoalEvaluation* ha l'obiettivo di valutare il singolo 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 cercando nella base di conoscenza del motore una clausola compatibile con tale subgoal: il motore transita quindi nello stato *RuleSelection*.

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

i. nello stato *GoalSelection* se la valutazione del predicato ha successo

ii. nello stato *Backtrack* se la valutazione del predicato fallisce

iii. 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

iv. 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 una `HaltException` lanciata dal predicato viene catturata dal primo blocco `catch` che fa transitare il motore nello stato *HALT*. Infine il secondo blocco `catch` cattura un'istanza di `Throwable` e serve appositamente per gestire le eccezioni eventualmente lanciate dal predicato. La gestione delle eccezioni era stata quindi prevista dagli sviluppatori di tuProlog, ma non ancora implementata: infatti attualmente nel `catch` tutto ciò che si fa è semplicemente stampare su standard error lo stacktrace dell'errore (che all'atto pratico sarà un'istanza di `PrologError`) e, come nel caso precedente, far transitare il motore 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`:

i. si effettua un cast dell'istanza di `Throwable` in un'istanza di `PrologError`

ii. come specificato dallo standard, il subgoal in cui si è verificato l'errore viene sostituito dal subgoal `throw(Error)`. La macchina ricava le informazioni sull'errore dall'istanza di `PrologError` catturata, e assegna il termine `throw/1` alla variabile `currentGoal` del corrente `ExecutionContext` del motore

iii. infine il motore transita nello stato *Exception*

5.4 La classe `StateException`

La classe `StateException` modella il nuovo stato *Exception* che ha il compito di gestire le eccezioni. Come per gli altri stati, il comportamento è definito nel metodo `doJob()` della classe.

Per gestire le eccezioni, la macchina controlla il goal che in questo moeffettua 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:

i. taglia tutti i punti di scelta generati da `Goal` (attraverso il metodo `cut()` della classe `EngineManager`)

ii. unifica l'argomento di `throw/1` con il secondo argomento di `catch/3` (attraverso il metodo `unify()` della classe `Term`)

iii. 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`)

Infine la macchina transita nello stato *GoalSelection* se è stato trovato un nodo `catch/3` appropriato e quindi l'errore può essere gestito, oppure nello stato *HALT* se tale nodo non è stato trovato: l'errore pertanto non può essere gestito e la macchina termina l'esecuzione.

5.5 `throw/1` e `catch/3`

L'implementazione dei due predicati `throw/1` e `catch/3` è molto semplice, 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 particolare dallo stato *Exception*.

Il predicato `throw(Error)` è implementato in Java e non fa altro che lanciare un'istanza di `PrologError` corrispondente all'errore `Error`. Come detto tale istanza sarà catturata nello stato *GoalEvaluation* e le informazioni sull'errore che si è verificato saranno poi utilizzate dallo stato *Exception* per ricercare un subgoal `catch/3` opportuno che sia in grado di gestirlo.

Il predicato `catch(Goal, Catcher, Handler)` è invece implementato in Prolog perché `Goal` potrebbe essere un predicato non-deterministico e in tal caso deve essere rieseguito attraverso il meccanismo del backtracking. Il predicato non fa altro che chiamare `Goal`, poi 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:

i. `call(Goal)` è vero, oppure

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

`throw/1` e `catch/3` dovrebbero teoricamente essere implementati come built-in, in modo da essere il più vicino possibili al motore: la loro collocazione dovrebbe quindi essere la classe `BuiltIn`. Tuttavia in tuProlog la libreria `BuiltIn` è gestita differentemente dalle altre librerie, e in particolar modo è gestita dal gestore dei predicati primitivi invece che dal gestore delle librerie. Il `PrimitiveManager` riesce però a gestire soltanto predicati, funtori e direttive scritti in linguaggio nativo (Java o C#), mentre ignora i predicati scritti in Prolog, come lo è `catch/3`. Al contrario il `LibraryManager` riesce a gestire entrambi i tipi di predicati. Pertanto una soluzione per inserire i due predicati appena implementati tra i built-in del motore potrebbe essere quella di far gestire anche `BuiltIn` dal `LibraryManager`. Tuttavia un'ulteriore problema che ci si trova ad affrontare è che la definizione dell'operatore `:-`, che è indispensabile per effettuare correttamente il parsing della clausola `catch/3`, si trova nella `BasicLibrary`: introducendo quindi nella classe `BuiltIn` una teoria che usa operatori non ancora dichiarati causa quindi un errore di parsing. Per questi motivi si è scelto di collocare temporaneamente i due predicati nella `BasicLibrary` (dove comunque si trovano predicati e funtori di base tipici di tutti i sistemi Prolog, come lo sono `throw/1` e `catch/3`), in attesa di sviluppi futuri riguardanti l'organizzazione dei diversi gestori di tuProlog.

5.6 Collaudo dell'estensione

Per quanto riguarda il collaudo dell'estensione, si è verificato che il funzionamento del motore non sia mutato nei confronti delle suite di test già esistenti. La verifica è andata a buon termine, e ciò era abbastanza prevedibile proprio per 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 di eccezioni, quindi la macchina non transita mai nel nuovo stato *Exception*.

Successivamente si è collaudato il funzionamento congiunto dei due nuovi predicati `throw(Error)` e `catch(Goal, Catcher, Handler)`, verificando che siano rispettati tutti i requisiti che lo standard impone per il comportamento dei due predicati. Tali requisiti sono elencati di seguito, e per ognuno di essi è riportato anche come la verifica è stata eseguita:

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

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

?- catch(p(0), E, atom_length(E, Length)).

yes.

E/error

Length/5
```

ii. deve essere eseguito il più vicino antenato `catch/3` nell'albero di risoluzione il cui secondo argomento unifica con `Error`

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

p(1).

?- catch(p(1), E, fail), catch(p(0), E, true).

yes.

E/error
```

iii. 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

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

?- catch(p(0), error(X), true).

no.
```

iv. `catch/3` deve fallire se `Handler` è falso

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

?- catch(p(0), E, false).

no.
```

v. 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

```
p(0).

p(1) :- throw(error).

p(2).

?- catch(p(X), E, true).

yes.

X/0

E/error

Next solution?

Yes.

E/error

X/1
```

```
Next solution?

no.
```

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

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

?- catch(p(0), E, throw(err)).

no.
```

Per il collaudo si è utilizzato lo strumento JUnit, e risulta che i predicati `throw/1` e `catch/3` rispettano tutti i requisiti di funzionamento specificati dallo standard.

6 Eccezioni nei predicati

Dopo aver modificato la macchina a stati finiti che è alla base del motore e implementato i due predicati standard `throw/1` e `catch/3`, il passo seguente 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.

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

Come discusso nel capitolo 4, ci sono 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) si riescono tipicamente a riscontrare solo durante l'esecuzione. Inoltre il controllo della presenza degli errori deve avvenire sia per i predicati implementati in Java, sia per quelli implementati in Prolog.

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

Pertanto in ogni libreria sono previste delle guardie apposite per i predicati Prolog, ovvero dei metodi Java 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 Java.

Nel seguito verrà analizzato il funzionamento di tutti (e soli) i predicati compresi nella distribuzione di tuProlog che possono presentare errori, suddivisi per libreria (`BuiltIn`, `BasicLibrary`, `ISOLibrary`, `DCGLibrary`, `IOLibrary`). Per ognuno di questi verranno descritte le eccezioni che può lanciare e in quali casi ciò può avvenire, e per ogni eccezione verrà descritta la sintassi precisa sia del termine `Error_term`, sia del termine `tuProlog_term`. Il primo termine è il termine standard, secondo quanto descritto nel capitolo 2, mentre il secondo termine è il termine non standard caratteristico dell'implementazione di tuProlog. Per quest'ultimo termine si è cercato di seguire lo stato dell'arte, e quindi la sua sintassi si ispira a quella del termine `SICStus_error` descritto nel capitolo 3. È da notare che nelle librerie sopra citate manca la `JavaLibrary`; questo perché si è scelto di trattare in modo diverso le eccezioni Java che si all'interno di programmi Prolog che utilizzano oggetti Java durante la computazione (e che quindi fanno uso dei predicati appartenenti alla `JavaLibrary`). Questo aspetto sarà discusso nel prossimo capitolo. Infine il capitolo si chiude con la descrizione del collaudo effettuato per verificare la correttezza di questo ulteriore passo dell'estensione del motore tuProlog per la gestione delle eccezioni.

6.1 `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 mai essere scaricati.

`asserta/1`

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

Eccezioni lanciate:

- `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 (`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`)

`assertz/1`

`assertz(Clause)` è vero, con l'effetto collaterale che la clausola `Clause` è aggiunta alla fine della base di conoscenza del motore.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Clause` è una variabile. `Goal` è il goal in cui si è verificato il problema (`assertz(Clause)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`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 (`assertz(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`)

`'$retract'/1`

`'$retract'(Clause)` è vero se la base di conoscenza del motore contiene almeno una clausola che unifica con `Clause`. Come effetto collaterale, la clausola è rimossa dalla base di conoscenza. Non è rieseguibile.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Clause` è una variabile. `Goal` è il goal in cui si è verificato il problema (`'$retract'(Clause)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`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 (`'$retract'(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`)

`abolish/1`

`abolish(PI)` è vero e rimuove dalla base di conoscenza del motore tutte le clausole corrispondenti all'indicatore di predicato `PI`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

se `PI` è una variabile. `Goal` è il goal in cui si è verificato il problema (`abolish(PI)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Clause` non è un indicatore di predicato. `Goal` è il goal in cui si è verificato il problema (`abolish(PI)`), `ArgNo` è indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `PI` (`predicate_indicator`), `Culprit` è il termine errato trovato (`PI`)

halt/1

`halt(X)` termina una dimostrazione Prolog, ritornando al sistema il valore `X` come messaggio.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `X` è una variabile. `Goal` è il goal in cui si è verificato il problema (`halt(X)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `X` non è un intero. `Goal` è il goal in cui si è verificato il problema (`halt(X)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `X` (`integer`), `Culprit` è il termine errato trovato (`X`)

load_library/1

`load_library(LibraryName)` è vero se `LibraryName` è il nome di una libreria tuProlog disponibile per essere caricata nel motore. Come

effetto collaterale, la libreria specificata viene caricata. `LibraryName` è il nome completo della classe Java che implementa la libreria.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `LibraryName` è una variabile. `Goal` è il goal in cui si è verificato il problema (`load_library(LibraryName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `LibraryName` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`load_library(LibraryName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `LibraryName` (`atom`), `Culprit` è il termine errato trovato (`LibraryName`)

- `error(existence_error(ObjectType, Culprit), existence_error(Goal, ArgNo, ObjectType, Culprit, Message))`

  se la libreria `LibraryName` non esiste. `Goal` è il goal in cui si è verificato il problema (`load_library(LibraryName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ObjectType` è il tipo dell'oggetto che non esiste (`class`), `Culprit` è il termine errato trovato (`LibraryName`), `Message` fornisce informazioni aggiuntive sull'errore che si è verificato.

`unload_library/1`

`unload_library(LibraryName)` è vero se `LibraryName` è il nome di una libreria correntemente caricata nel motore. Come effetto collaterale, la libreria specificata viene scaricata. `LibraryName` è il nome completo della classe Java che implementa la libreria.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `LibraryName` è una variabile. `Goal` è il goal in cui si è verificato il problema (`unload_library(LibraryName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `LibraryName` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`unload_library(LibraryName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `LibraryName` (`atom`), `Culprit` è il termine errato trovato (`LibraryName`)

- `error(existence_error(ObjectType, Culprit), existence_error(Goal, ArgNo, ObjectType, Culprit, Message))`

  se la libreria `LibraryName` non esiste. `Goal` è il goal in cui si è verificato il problema (`unload_library(LibraryName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ObjectType` è il tipo dell'oggetto che non esiste (`class`), `Culprit` è il termine errato trovato (`LibraryName`), `Message` fornisce informazioni aggiuntive sull'errore che si è verificato.

`'$call'/1`

`'$call'(G)` è vero se e solo se `Goal` rappresenta un goal che è vero. Non è opaco al cut.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

se `G` è una variabile. `Goal` è il goal in cui si è verificato il problema (`'$call'(G)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `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 (`'$call'(G)`), `ArgNo` indica quale argomento ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `G` (`callable`), `Culprit` è il termine errato trovato (`G`)

is/2

`is(X, Y)` è vero se e solo se `X` è unificabile con il valore dell'espressione `Y`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Y` è una variabile. `Goal` è il goal in cui si è verificato il problema (`is(X, Y)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Y` non è un'espressione valutabile correttamente. `Goal` è il goal in cui si è verificato il problema (`is(X, Y)`), `ArgNo` indica quale argomento ha causato il problema (`2`), `ValidType` è il tipo di dato previsto per `Y` (`evaluable`), `Culprit` è il termine errato trovato (`Y`)

- `error(evaluation_error (Error), evaluation_error(Goal, ArgNo, Error))`

se si verifica un errore durante la valutazione di `Y`. `Goal` è il goal in cui si è verificato il problema (`is(X, Y)`), `ArgNo` indica quale argomento ha causato il problema (`2`), `Error` è l'errore che si è verificato (ad esempio `zero_divisor` nel caso di divisione per zero).

`'$tolist'/2`

`'$tolist'(Struct, List)` è vero se `Struct` è una struttura (un atomo, una lista o un termine composto) e `List` è la rappresentazione sotto forma di lista del termine `Struct` (il primo elemento della lista è dato dal funtore della struttura, mentre tutti gli altri elementi sono gli argomenti della struttura).

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Struct` è una variabile. `Goal` è il goal in cui si è verificato il problema (`'$tolist'(Struct, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Struct` non è una struttura. `Goal` è il goal in cui si è verificato il problema (`'$tolist'(Struct, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `Struct` (`struct`), `Culprit` è il termine errato trovato (`Struct`)

`'$fromlist'/2`

`'$fromlist'(Struct, List)` è vero se la struttura `Struct` unifica con la sua rappresentazione sotto forma di lista `List`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `List` è una variabile. `Goal` è il goal in cui si è verificato il problema (`'$fromlist'(Struct, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `List` non è una lista. `Goal` è il goal in cui si è verificato il problema (`'$fromlist'(Struct, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`), `ValidType` è il tipo di dato previsto per `List` (`list`), `Culprit` è il termine errato trovato (`List`)

`'$append'/2`

`'$append'(Element, List)` è vero se `List` è una lista, con l'effetto collaterale che il termine `Element` è aggiunto alla lista.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `List` è una variabile. `Goal` è il goal in cui si è verificato il problema (`'$append'(Element, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `List` non è una lista. `Goal` è il goal in cui si è verificato il problema (`'$append'(Element, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`), `ValidType` è il tipo di dato previsto per `List` (`list`), `Culprit` è il termine errato trovato (`List`)

'$find'/2

'$find'(Clause, ClauseList) è vero se ClauseList è una lista e Clause è una clausola, con l'effetto collaterale che tutte le clausole della base di conoscenza del motore che fanno match con Clause sono aggiunte alla lista.

Eccezioni lanciate:

- error(instantiation_error, instantiation_error(Goal, ArgNo))

  se Clause è una variabile. Goal è il goal in cui si è verificato il problema ('$find'(Clause, ClauseList)), ArgNo indica quale argomento del predicato ha causato il problema (1)

- error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))

  se ClauseList non è una lista. Goal è il goal in cui si è verificato il problema ($find'(Clause, ClauseList)), ArgNo indica quale argomento del predicato ha causato il problema (2), ValidType è il tipo di dato previsto per Clause (list), Culprit è il termine errato trovato (ClauseList)

get_prolog_flag/2

get_prolog_flag(Flag, Value) è vero se e solo se Flag è un flag supportato dal motore e Value è il valore correntemente associato a Flag. Non è rieseguibile.

Eccezioni lanciate:

- error(instantiation_error, instantiation_error(Goal, ArgNo))

  se Flag è una variabile. Goal è il goal in cui si è verificato il problema (get_prolog_flag(Flag, Value)), ArgNo indica quale argomento del predicato ha causato il problema (1)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Flag` non è una struttura. `Goal` è il goal in cui si è verificato il problema (`get_prolog_flag(Flag, Value)`), `ArgNo` indica quale argomento ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `Flag` (`struct`), `Culprit` è il termine errato trovato (`Flag`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)`

  se `Flag` non è definito nel motore. `Goal` è il goal in cui si è verificato il problema (`get_prolog_flag(Flag, Value)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidDomain` è il dominio previsto per `Flag` (`prolog_flag`), `Culprit` è il termine errato trovato (`Flag`)

`set_prolog_flag/2`

`set_prolog_flag(Flag, Value)` è vero, e come effetto collaterale associa `Value` al flag `Flag`. `Value` deve essere un valore tra quelli ammissibili per `Flag`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Flag` o `Value` sono variabili. `Goal` è il goal in cui si è verificato il problema (`set_prolog_flag(Flag, Value)`), `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 `Flag` non è una struttura oppure `Value` non è un termine "ground". `Goal` è il goal in cui si è verificato il problema

(`set_prolog_flag(Flag, Value)`), `ArgNo` indica quale argomento ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `Flag` o `Value` (`struct` o `ground`), `Culprit` è il termine errato trovato (`Flag` o `Value`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)`

  se `Flag` non è definito nel motore oppure `Value` non è ammissibile per `Flag`. `Goal` è il goal in cui si è verificato il problema (`set_prolog_flag(Flag, Value)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidDomain` è il dominio previsto per `Flag` o `Value` (`prolog_flag` o `flag_value`), `Culprit` è il termine errato trovato (`Flag` o `Value`)

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se `Flag` non è modificabile. `Goal` è il goal in cui si è verificato il problema (`set_prolog_flag(Flag, Value)`), `Operation` è l'operazione che si cerca di eseguire (`modify`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`flag`), `Culprit` è l'oggetto su cui si cerca di eseguire l'operazione (`Flag`), `Message` fornisce informazioni aggiuntive sull'errore che si è verificato (quando non ce ne sono, come in questo caso, per convenzione si usa l'intero `0`).

`'$op'/3`

`'$op'(Priority, Specifier, Operator)` è sempre vero, e come effetto collaterale modifica la tabella degli operatori del motore, aggiungendo alla tabella l'operatore `Operator` con priorità `Priority` (più è basso il valore di `Priority`, più l'operatore è prioritario) e associatività determinata da `Specifier`. Se nella tabella degli operatori

esiste già un operatore con il simbolo `Operator` e con associatività `Specifier`, allora il predicato modifica la sua priorità in accordo al valore specificato di `Priority`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Priority` o `Specifier` o `Operator` sono variabili. `Goal` è il goal in cui si è verificato il problema (`'$op'(Priority, Specifier, Operator)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2` o `3`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Priority` non è un intero oppure `Specifier` non è un atomo oppure `Operator` non è un atomo o una lista di atomi. `Goal` è il goal in cui si è verificato il problema (`'$op'(Priority, Specifier, Operator)`), `ArgNo` indica quale argomento ha causato il problema (`1` o `2` o `3`), `ValidType` è il tipo di dato previsto per `Priority` o `Specifier` o `Operator` (`integer` o `atom` o `atom_or_atom_list`), `Culprit` è il termine errato trovato (`Priority` o `Specifier` o `Operator`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)`

  se `Priority` o `Specifier` sono del tipo di dato corretto, ma presentano valori non ammissibili per la priorità o l'associatività di un operatore. `Goal` è il goal in cui si è verificato il problema (`'$op'(Priority, Specifier, Operator)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidDomain` è il dominio previsto per `Priority` o `Specifier`

(`operator_priority` o `operator_specifier`), `Culprit` è il termine errato trovato (`Priority` o `Specifier`)

6.2 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.

text_concat/3

`text_concat(TextSource1, TextSource2, TextDest)` è vero se e solo se `TextDest` è il testo che risulta dalla concatenazione di `TextSource1` con `TextSource2`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `TextSource1` o `TextSource2` sono variabili. `Goal` è il goal in cui si è verificato il problema (`text_concat(TextSource1, TextSource2, TextDest)`), `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 `TextSource1` o `TextSource2` non sono atomi. `Goal` è il goal in cui si è verificato il problema (`text_concat(TextSource1, TextSource2, TextDest)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `TextSource1` e `TextSource2` (`atom`), `Culprit` è il termine errato trovato (`TextSource1` o `TextSource2`)

num_atom/2

`num_atom(Number, Atom)` ha successo se e solo se `Atom` è la rappresentazione sotto forma di atomo (stringa) del numero `Number`.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Atom` è una variabile e `Number` non è un numero oppure se `Atom` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`num_atom(Number, Atom)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `Number` o `Atom` (`number` o `atom`), `Culprit` è il termine errato trovato (`Number` o `Atom`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)`

  se Atom è un atomo che però non rappresenta alcun numero. `Goal` è il goal in cui si è verificato il problema (`num_atom(Number, Atom)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`), `ValidDomain` è il dominio previsto per `Atom` (`num_atom`), `Culprit` è il termine errato trovato (`Atom`)

`set_theory/1`

`set_theory(TheoryText)` è vero se e solo se `TheoryText` è la rappresentazione testuale di una teoria tuProlog corretta, con l'effetto collaterale di settarla come nuova teoria del motore.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `TheoryText` è una variabile. `Goal` è il goal in cui si è verificato il problema (`set_theory(TheoryText)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

se `TheoryText` non è un atomo (stringa). `Goal` è il goal in cui si è verificato il problema (`set_theory(TheoryText)`), `ArgNo` indica quale argomento ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `TheoryText` (`atom`), `Culprit` è il termine errato trovato (`TheoryText`)

- `error(syntax_error (Message), syntax_error(Goal, Line, Position, Message))`

  se `TheoryText` non è una teoria valida. `Goal` è il goal in cui si è verificato il problema (`set_theory(TheoryText)`), `Message` è una stringa che descrive l'errore che si è verificato, `Line` e `Position` identificano rispettivamente la linea e la posizione di `TheoryText` in cui su è verificato l'errore (se il motore non è in grado di fornire tali informazioni i due termini valgono `-1`)

`add_theory/1`

`add_theory(TheoryText)` è vero se e solo se `TheoryText` è la rappresentazione testuale di una teoria tuProlog corretta, con l'effetto collaterale di aggiungerla alla corrente teoria del motore.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `TheoryText` è una variabile. `Goal` è il goal in cui si è verificato il problema (`add_theory(TheoryText)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `TheoryText` non è un atomo (stringa). `Goal` è il goal in cui si è verificato il problema (`add_theory(TheoryText)`), `ArgNo` indica quale argomento ha causato il problema (`1`), `ValidType` è il tipo di

dato previsto per `TheoryText` (`atom`), `Culprit` è il termine errato trovato (`TheoryText`)

- `error(syntax_error (Message), syntax_error(Goal, Line, Position, Message))`

  se `TheoryText` non è una teoria valida. `Goal` è il goal in cui si è verificato il problema (`add_theory(TheoryText)`), `Message` è una stringa che descrive l'errore che si è verificato, `Line` e `Position` identificano rispettivamente la linea e la posizione di `TheoryText` in cui su è verificato l'errore

agent/1

`agent(TheoryText)` è vero, e fa partire l'esecuzione di un agente tuProlog la cui base di conoscenza è fornita sotto forma di testo dal termine `TheoryText`. Il goal è descritto nella base di conoscenza.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `TheoryText` è una variabile. `Goal` è il goal in cui si è verificato il problema (`agent(TheoryText)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `TheoryText` non è un atomo (stringa). `Goal` è il goal in cui si è verificato il problema (`agent(TheoryText)`), `ArgNo` indica quale argomento ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `TheoryText` (`atom`), `Culprit` è il termine errato trovato (`TheoryText`)

agent/2

`agent(TheoryText, G)` è vero, e fa partire l'esecuzione di un agente

tuProlog la cui base di conoscenza è fornita sotto forma di testo dal termine `TheoryText` che risolve la query `G`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `TheoryText` o `G` sono variabili. `Goal` è il goal in cui si è verificato il problema (`agent(TheoryText, G)`), `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 `TheoryText` non è un atomo oppure `G` non è una struttura. `Goal` è il goal in cui si è verificato il problema (`agent(TheoryText, G)`), `ArgNo` indica quale argomento ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `TheoryText` o `G` (`atom` o `struct`), `Culprit` è il termine errato trovato (`TheoryText` o `G`)

Confronto di espressioni:

(template generico: *`pred`*`(@expr, @expr)`)

`'=:='` (uguale), `'=\='` (diverso), `'>'` (maggiore), `'<'` (minore), `'>='` (maggiore o uguale), `'=<'` (minore o uguale)

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se uno dei due argomenti è una variabile. `Goal` è il goal in cui si è verificato il problema, `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 uno dei due argomenti non è un'espressione valutabile correttamente. `Goal` è il goal in cui si è verificato il problema, `ArgNo` indica quale argomento ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per l'argomento errato (`evaluable`), `Culprit` è il termine errato trovato.

- `error(evaluation_error (Error), evaluation_error(Goal, ArgNo, Error))`

  se si verifica un errore durante la valutazione di uno dei due argomenti. `Goal` è il goal in cui si è verificato il problema, `ArgNo` indica quale argomento ha causato il problema (`1` o `2`), `Error` è l'errore che si è verificato (ad esempio `zero_divisor` nel caso di divisione per zero).

`arg/3`

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

Eccezioni lanciate:

- `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/2

`clause(Head, Body)` è vero se e solo se nella base di conoscenza del motore è presente un predicato la cui testa fa match con `Head` e il cui corpo fa match con `Body` (il corpo di un fatto è l'atomo `true`). `Head` deve essere almeno parzialmente instanziato.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Head` è una variabile. `Goal` è il goal in cui si è verificato il problema (`clause_guard(Head, Body)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

call/1

Lancia le stesse eccezioni di `'$call'/1`, con la sola differenza che lo specifico goal che lancia l'eccezione non è `'$call'(G)`, bensì `call_guard(G)`.

member/2

`member(Element, List)` è vero se e solo se `Element` è un elemento della lista `List`.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `List` non è una lista. `Goal` è il goal in cui si è verificato il problema (`member_guard(Element, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`2`), `ValidType` è il tipo di dato previsto per `List` (`list`), `Culprit` è il termine errato trovato (`List`)

`reverse/2`

`reverse(List, ReversedList)` è vero se e solo se `ReversedList` è la lista invertita di `List`.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `List` non è una lista. `Goal` è il goal in cui si è verificato il problema (`reverse_guard(List, ReversedList)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `List` (`list`), `Culprit` è il termine errato trovato (`List`)

`element/3`

`element(Position, List, Element)` è vero se e solo se `Element` è l'elemento in posizione `Position` nella lista `List` (il conteggio parte da 1).

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `List` non è una lista. `Goal` è il goal in cui si è verificato il problema (`element_guard(Position, List, Element)`), `ArgNo` indica quale argomento del predicato ha causato il problema,

(2), `ValidType` è il tipo di dato previsto per l'argomento errato (`list`), `Culprit` è il termine errato trovato (`List`)

`delete/3`

`delete(Element, ListSource, ListDest)` è vero se e solo se la lista `ListDest` può essere ottenuta rimuovendo l'elemento `Element` dalla lista `ListSource`.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `ListSource` non è una lista. `Goal` è il goal in cui si è verificato il problema (`delete_guard(Element, ListSource, ListDest)`), `ArgNo` indica quale argomento del predicato ha causato il problema, (2), `ValidType` è il tipo di dato previsto per `ListSource` (`list`), `Culprit` è il termine errato trovato (`ListSource`)

`assert/1`

Lancia le stesse eccezioni di `assertz/1`.

`retract/1`, **`retractall/1`**

Lanciano le stesse eccezioni di `'$retract'/1`, con la sola differenza che lo specifico goal che lancia l'eccezione non è `'$retract'(Clause)`, bensì `retract_guard(Clause)`.

`findall/3`, **`setof/3`**, **`bagof/3`**

I tre predicati `findall(Template, G, List)`, `bagof(Template, G, Instances)` e `setof(Template, G, List)` servono per ricercare tutte le soluzioni associate al goal `G`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `G` è una variabile. `Goal` è il goal in cui si è verificato il problema, `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 (ad esempio è un numero). `Goal` è il goal in cui si è verificato il problema, `ArgNo` indica quale argomento del predicato ha causato il problema (`2`), `ValidType` è il tipo di dato previsto per `G` (`callable`), `Culprit` è il termine errato trovato (`G`)

6.3 `ISOLibrary`

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

`atom_length/2`

`atom_length(Atom, Length)` è vero se e solo se l'intero `Length` corrisponde al numero di caratteri dell'atomo `Atom`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Atom` è una variabile. `Goal` è il goal in cui si è verificato il problema (`atom_length(Atom, Length)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Atom` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`atom_length(Atom, Length)`), `ArgNo` indica quale

argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `Atom` (`atom`), `Culprit` è il termine errato trovato (`Atom`)

`atom_chars/2`

`atom_chars(Atom, List)` ha successo se e solo se `List` è una lista i cui elementi sono i caratteri (in ordine) che compongono `Atom`.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Atom` è una variabile e `List` non è una lista oppure se `List` è una variabile e `Atom` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`atom_chars(Atom, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `Atom` o `List` (`atom` o `list`), `Culprit` è il termine errato trovato (`Atom` o `List`)

`char_code/2`

`char_code(Char, Code)` ha successo se e solo se `Code` è l'intero corrispondente al carattere `Char`.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Code` è una variabile e `Char` non è un carattere (atomo di lunghezza 1) oppure se `Char` è una variabile e `Code` non è un intero. `Goal` è il goal in cui si è verificato il problema (`char_code(Char, Code)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1` o `2`), `ValidType` è il tipo di dato previsto per `Char` o `Code` (`character` o `integer`), `Culprit` è il termine errato trovato (`Char` o `Code`)

sub_atom/5

`sub_atom(Atom, Before, Length, After, SubAtom)` è vero se e solo se `SubAtom` è il sotto atomo di `Atom` di lunghezza `Length` che ha `Before` caratteri prima e `After` caratteri dopo. È rieseguibile.

Eccezioni lanciate:

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Atom` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`sub_atom_guard(, Before, Length, After, SubAtom)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `Atom` (`atom`), `Culprit` è il termine errato trovato (`Atom`)

6.4 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.

Il programma Prolog di Figura 6 è un esempio di DCG che esprime il concetto che una frase è composta da una parte verbale e una parte nominale. La parte verbale a sua volta è composta da un verbo (che può essere solo `loves`) e da una parte nominale, che può essere `charles` o `linda`.

Si noti che la sequenza `-->` serve a separare la testa dal corpo di ogni regola della grammatica, mentre la differenza tra i simboli non terminali della grammatica e quelli terminali è che i primi sono rappresentati come atomi, mentre i secondi come una lista. Quindi una lista Prolog rappresenta una sequenza di uno o più simboli terminali della grammatica, mentre la lista vuota rappresenta la sequenza vuota.

```
sentence --> noun_phrase, verb_phrase

verb_phrase --> verb, noun_phrase

noun_phrase --> [charles]

noun_phrase --> [linda]

verb --> [loves]
```

Figura 6: Esempio di DCG in Prolog [1]

Detto questo, si può chiedere al motore se la frase "`charles loves linda`" può essere derivata dallo scopo della grammatica (che in questo caso è `sentence`) eseguendo la query:

```
?- phrase(sentence, [charles, loves, linda]).
```

Invece la query:

```
?- phrase(sentence, [Who, loves, linda]).
```

in base alla grammatica specificata, produce due soluzioni: `Who/charles` e `Who/linda`.

`phrase/2`

`phrase(Category, List)` è vero se e solo se la lista `List` corrisponde a una frase (sequenza di simboli non terminali) derivabile da `Category`. `Category` può essere un qualsiasi simbolo non terminale della grammatica e al momento della chiamata deve essere instanziato a un termine non variabile. Se al momento della chiamata `List` è legato a una lista di simboli non terminali, allora il goal consiste nel verificare che la frase espressa da `List` sia di tipo `Category`; altrimenti, se `List` non è legato, la grammatica viene usata per scopi generativi.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

se `Category` è una variabile. `Goal` è il goal in cui si è verificato il problema (`phrase_guard(Category, List)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

`phrase/3`

`phrase(Category, List, Rest)` è vero se e solo se il segmento compreso tra l'inizio della lista `List` e l'inizio della lista `Rest` corrisponde a una frase (sequenza di simboli non terminali) derivabile da `Category`. Di nuovo, `Category` può essere un qualsiasi simbolo non terminale della grammatica e al momento della chiamata deve essere instanziato a un termine non variabile.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Category` è una variabile. `Goal` è il goal in cui si è verificato il problema (`phrase_guard(Category, List, Rest)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

6.5 `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.

`see/1`

`see(StreamName)` è usato per creare/aprire uno stream di input; il predicato è vero se e solo se `StreamName` è una stringa rappresentante il nome di un file da essere creato o acceduto come stream di input, oppure se è la stringa stdin che seleziona lo standard input come stream di input.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `StreamName` è una variabile. `Goal` è il goal in cui si è verificato il problema (`see(StreamName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `StreamName` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`see(StreamName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `StreamName` (`atom`), `Culprit` è il termine errato trovato (`StreamName`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)`

  se `StreamName` non è il nome di un file accessibile. `Goal` è il goal in cui si è verificato il problema (`see(StreamName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidDomain` è il dominio previsto per `StreamName` (`stream`), `Culprit` è il termine errato trovato (`StreamName`)

`tell/1`

`tell(StreamName)` è usato per creare/aprire uno stream di output; il predicato è vero se e solo se `StreamName` è una stringa rappresentante il nome di un file da essere creato o acceduto come stream di output, oppure se è la stringa stdout che seleziona lo standard output come stream di output.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

se `StreamName` è una variabile. `Goal` è il goal in cui si è verificato il problema (`tell(StreamName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `StreamName` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`tell(StreamName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `StreamName` (`atom`), `Culprit` è il termine errato trovato (`StreamName`)

- `error(domain_error (ValidDomain, Culprit), domain_error(Goal, ArgNo, ValidDomain, Culprit)`

  se `StreamName` non è il nome di un file accessibile. `Goal` è il goal in cui si è verificato il problema (`tell(StreamName)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidDomain` è il dominio previsto per `StreamName` (`stream`), `Culprit` è il termine errato trovato (`StreamName`)

`get0/1`

`get0(Value)` è vero se e solo se `Value` è il prossimo carattere (il cui codice può spaziare nell'intero range ASCII) disponibile dallo stream di input, oppure -1 se non ci sono caratteri disponibili; come effetto collaterale il carattere è rimosso dallo stream di input.

Eccezioni lanciate:

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile leggere dallo stream di input. `Goal` è il goal in cui si è verificato il problema (`get0(Value)`), `Operation` è l'operazione che si cerca di eseguire (`input`), `ObjectType` è il tipo

dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di input, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

`get/1`

`get(Value)` è vero se e solo se `Value` è il prossimo carattere (il cui codice può spaziare nel range 32..255 del codice ASCII) disponibile dallo stream di input, oppure -1 se non ci sono caratteri disponibili; come effetto collaterale il carattere (insieme a tutti quelli che lo precedono che non sono compresi nel range 32..255) è rimosso dallo stream di input.

Eccezioni lanciate:

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile leggere dallo stream di input. `Goal` è il goal in cui si è verificato il problema (`get(Value)`), `Operation` è l'operazione che si cerca di eseguire (`input`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di input, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

`put/1`

`put(Char)` scrive il carattere `Char` sullo stream di output; è vero se e solo se l'operazione è possibile.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Char` è una variabile. `Goal` è il goal in cui si è verificato il problema (`put(Char)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `Char` non è un carattere (atomo di lunghezza 1). `Goal` è il goal in cui si è verificato il problema (`put(Char)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `Char` (`char`), `Culprit` è il termine errato trovato (`Char`)

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile scrivere sullo stream di output. `Goal` è il goal in cui si è verificato il problema (`put(Char)`), `Operation` è l'operazione che si cerca di eseguire (`output`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di output, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

`tab/1`

`tab(NumSpaces)` scrive `NumSpaces` spazi (carattere ASCII 32) sullo stream di output; il predicato è vero se e solo se l'operazione è possibile.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `NumSpaces` è una variabile. `Goal` è il goal in cui si è verificato il problema (`tab(NumSpaces)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

se `NumSpaces` non è un intero. `Goal` è il goal in cui si è verificato il problema (`tab(NumSpaces)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `NumSpaces` (`integer`), `Culprit` è il termine errato trovato (`NumSpaces`)

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile scrivere sullo stream di output. `Goal` è il goal in cui si è verificato il problema (`tab(NumSpaces)`), `Operation` è l'operazione che si cerca di eseguire (`output`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di output, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

`write/1`

`write(Term)` scrive il termine `Term` sullo stream di output corrente; il predicato è vero se e solo se l'operazione è possibile.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Term` è una variabile. `Goal` è il goal in cui si è verificato il problema (`write(Term)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile scrivere sullo stream di output. `Goal` è il goal in cui si è verificato il problema (`write(Term)`), `Operation` è l'operazione che si cerca di eseguire (`output`), `ObjectType` è il

tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di output, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

`print/1`

`print(Term)` scrive il termine `Term` sullo stream di output corrente, rimuovendo gli apici se il termine rappresenta una stringa; il predicato è vero se e solo se l'operazione è possibile.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `Term` è una variabile. `Goal` è il goal in cui si è verificato il problema (`print (Term)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile scrivere sullo stream di output. `Goal` è il goal in cui si è verificato il problema (`print (Term)`), `Operation` è l'operazione che si cerca di eseguire (`output`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di output, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

`read/1`

`read(Term)` è vero se e solo se `Term` è un termine Prolog disponibile sullo stream di input. Il termine deve terminare con il carattere `.`; come effetto collaterale il termine è rimosso dallo stream di input.

Eccezioni lanciate:

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se si è verificato un errore durante la lettura dallo stream di input. `Goal` è il goal in cui si è verificato il problema (`read(Term)`), `Operation` è l'operazione che si cerca di eseguire (`input`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di input, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

- `error(syntax_error (Message), syntax_error(Goal, Line, Position, Message))`

  se si è verificato un errore di sintassi durante la lettura dallo stream di input. `Goal` è il goal in cui si è verificato il problema (`read(Term)`), `Message` è la stringa letta dallo stream di input che ha generato l'errore di sintassi, `Line` e `Position` in questo caso valgono -1

`nl/0`

`nl` scrive un carattere di newline sullo stream di output; il predicato è vero se e solo se l'operazione è possibile.

Eccezioni lanciate:

- `error(permission_error (Operation, ObjectType, Culprit), permission_error(Goal, Operation, ObjectType, Culprit, Message))`

  se non è stato possibile scrivere sullo stream di output. `Goal` è il goal in cui si è verificato il problema (`nl`), `Operation` è l'operazione che si cerca di eseguire (`output`), `ObjectType` è il tipo dell'oggetto su cui si cerca di eseguire l'operazione (`stream`), `Culprit` è il nome dello stream di output, `Message` fornisce informazioni aggiuntive sull'errore che si è verificato

text_from_file/2

`text_from_file(File, Text)` è vero se e solo se `Text` è il testo contenuto nel file di nome `File`.

Eccezioni lanciate:

- `error(instantiation_error, instantiation_error(Goal, ArgNo))`

  se `File` è una variabile. `Goal` è il goal in cui si è verificato il problema (`text_from_file(File, Text)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`)

- `error(type_error (ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))`

  se `File` non è un atomo. `Goal` è il goal in cui si è verificato il problema (`text_from_file(File, Text)`), `ArgNo` indica quale argomento ha causato il problema (`1`), `ValidType` è il tipo di dato previsto per `File` (`atom`), `Culprit` è il termine errato trovato (`File`)

- `error(existence_error(ObjectType, Culprit), existence_error(Goal, ArgNo, ObjectType, Culprit, Message))`

  se il file `File` non esiste. `Goal` è il goal in cui si è verificato il problema (`text_from_file(File, Text)`), `ArgNo` indica quale argomento del predicato ha causato il problema (`1`), `ObjectType` è il tipo dell'oggetto che non esiste (`stream`), `Culprit` è il termine errato trovato (`File`), `Message` fornisce informazioni aggiuntive sull'errore che si è verificato (`File not found.`).

agent_file/1

`agent(TheoryFileName)` è vero se e solo se `TheoryFileName` è un file accessibile contenente una base di conoscenza Prolog, e come effetto

collaterale fa partire l'esecuzione di un agente tuProlog fornito di tale base di conoscenza.

Eccezioni lanciate:

Utilizza `text_from_file(File, Text)` con `TheoryFileName = File`, quindi lancia le stesse eccezioni

`solve_file/2`

`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.

Eccezioni lanciate:

- `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

`agent_file/1`

`consult(TheoryFileName)` è vero se e solo se `TheoryFileName` è un

file accessibile contenente una base di conoscenza Prolog, e come effetto collaterale consulta tale base di conoscenza e la aggiunge al motore.

Eccezioni lanciate:

Utilizza `text_from_file(File, Text)` con `TheoryFileName = File`, quindi lancia le stesse eccezioni. Inoltre lancia:

- `error(syntax_error (Message), syntax_error(Goal, Line, Position, Message))`

  se la teoria contenuta in `TheoryFileName` non è una teoria valida. `Goal` è il goal in cui si è verificato il problema (`add_theory(TheoryText)`), `Message` è una stringa che descrive l'errore che si è verificato, `Line` e `Position` identificano rispettivamente la linea e la posizione di `TheoryText` in cui su è verificato l'errore

6.6 Note implementative

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

Nell'implementazione si è cercato di individuare il maggior numero possibile di condizioni di errore, rispettando però sempre il requisito fondamentale di correttezza: se una chiamata a un predicato non falliva prima dell'introduzione del meccanismo delle eccezioni, non deve fallire neanche ora. In altre parole, il lancio di una eccezione deve avvenire soltanto in circostanze in cui il motore tuProlog originario fallisce. La correttezza del motore è garantita anche se ci si è dimenticati di identificare qualche condizione di errore inaspettata: in questo caso infatti il motore non lancia un'eccezione, ma comunque fallisce. In definitiva tale

estensione permette ad un utente di gestire gli errori che si possono verificare durante l'esecuzione, ma permette anche di non gestirli: in questo caso l'esecuzione fallirà, e l'estensione è come se non ci fosse.

Di nuovo, la malleabilità del motore ha permesso di implementare il lancio delle eccezioni modificando quasi esclusivamente le cinque librerie (`BuiltIn`, `BasicLibrary`, `ISOLibrary`, `DCGLibrary`, `IOLibrary`). Tuttavia sono state necessarie anche altre piccole modifiche al motore:

- alla classe `alice.tuprolog.FlagManager` sono stati aggiunti due metodi per ricavare informazioni su un flag:
  - `boolean isModifiable(String name)`

    restituisce `true` se esiste nel motore un flag di nome `name`, e tale flag è modificabile
  - `boolean isValidValue(String name, Term Value)`

    restituisce `true` se esiste nel motore un flag di nome `name`, e `Value` è un valore ammissibile per tale flag
- nella classe `alice.tuprolog.Prolog` il metodo `EngineManager getEngineManager()` è ora pubblico (in precedenza la visibilità era di package); in questo modo le varie librerie sono in grado di ricavare dal motore l'informazione sul goal correntemente in esecuzione e inserire tale informazione nell'eccezione lanciata
- nella classe `alice.tuprolog.PrimitiveInfo` il metodo `public Term evalAsFunctor(Struct g)` lancia ora un'istanza di `Throwable` in caso di errore durante la valutazione di `g`, mentre in precedenza ritornava `null`. Questo permette di discriminare il tipo di errore che si è verificato durante la valutazione di un funtore
- nella classe `alice.tuprolog.Library` il metodo `Term evalExpression(Term term)` rilancia l'istanza di `Throwable`

ricevuta dal metodo `evalAsFuntor()` della classe `alice.tuprolog.PrimitiveInfo`, sempre per permettere di discriminare il tipo di errore che si è verificato durante la valutazione di un funtore

6.7 Collaudo

Anche per il collaudo di questa parte dell'estensione si è provveduto innanzitutto a verificare che il funzionamento del motore non sia mutato nei confronti delle suite di test già esistenti. La verifica è andata a buon termine, perché come è stato detto l'introduzione del meccanismo di gestione degli errori non modifica la correttezza del motore. Quindi se un predicato falliva prima dell'implementazione delle eccezioni, continua ancora a fallire se tali eccezioni non sono gestite, come effettivamente avviene nelle suite di test esistenti.

Successivamente si è collaudato il lancio di tutte le possibili eccezioni che i predicati di libreria possono ora generare, provando ad invocare un predicato con dei parametri errati e verificando che il motore lanci l'eccezione corretta. Ad esempio, per verificare che `'$call'(X)` lancia un errore di istanziazione:

i. si è eseguito il predicato all'interno di un blocco `catch/3` che catturi tale tipo di errore

ii. si è verificato che la sintassi dell'errore lanciato è corretta in base all'invocazione. Nell'esempio l'errore che si viene a generare è:

```
error(instantiation_error,

instantiation_error('$call'(X),1))
```

Il collaudo ha avuto positivo per tutti i test eseguiti (che sono 50 per la libreria `BuiltIn`, 85 per la `BasicLibrary`, 7 per la `ISOLibrary`, 2 per la `DCGLibrary` e 27 per la `IOLibrary`).

7 Eccezioni Java in Prolog

L'ultimo problema da affrontare, strettamente legato al supporto alla programmazione multi-paradigma Java/Prolog offerto da tuProlog, è la generazione e la gestione di eccezioni Java all'interno di programmi Prolog che utilizzano oggetti Java durante la computazione.

Il capitolo inizia quindi con una introduzione sui meccanismi presenti in tuProlog per accedere al mondo Java da programmi Prolog, e descrive anche alcuni esempi di utilizzo dei predicati della `JavaLibrary`. Tale introduzione servirà per comprendere meglio l'ulteriore estensione del motore (presentata nella seconda parte del capitolo) che permette di lanciare e gestire eccezioni Java all'interno di programmi Prolog.

7.1 Accedere a Java da tuProlog

Uno dei maggiori vantaggi dell'architettura di tuProlog è che qualsiasi oggetto, classe o package Java può essere acceduto direttamente da un programma Prolog attraverso l'utilizzo dei predicati della `JavaLibrary`. In questo modo ad esempio è possibile accrescere le funzionalità di interazione di tuProlog, essendo possibile l'utilizzo di package come Swing, JDBC o RMI. Tutto questo è disponibile mediante una sola libreria, la `JavaLibrary` appunto.

Per consentire una corretta interazione tra Java e Prolog, tuProlog prevede un mapping completo e bidirezionale tra i tipi di dato dei due mondi:

- gli interi di tuProlog sono mappati in modo appropriato sui tipi di dato `int` o `long` di Java
- i tipi di dato `byte` e `short` di Java sono mappati su istanze della classe `Int` di tuProlog
- il tipo di dato `double` di Java è usato per mappare i numeri reali di tuProlog; se l'invocazione di un metodo Java ritorna un valore

`float`, questo valore viene trattato in modo appropriato e non c'è nessuna perdita di informazione

- i booleani di Java sono mappati su termini tuProlog costanti
- i `char` di Java sono mappati su atomi Prolog, mentre gli atomi sono mappati sul tipo di dato `String` di Java
- la variabile *any* `(_)` può essere utilizzata per specificare il valore null di Java

7.2 I predicati della `JavaLibrary`

La `JavaLibrary` offre i seguenti predicati:

`java_object/3`

`java_object(ClassName, ArgList, ObjId)` è vero se e solo se `ClassName` è il nome completo di una classe Java disponibile nel file system locale (es. `'Counter'` o `'java.io.FileInputStream'`), `ArgList` è una lista di argomenti che può essere utilizzata per istanziare un oggetto di tale classe (corrisponde agli argomenti del costruttore; la lista vuota fa match con il costruttore di default, mentre se si vuole istanziare un array di oggetti di un certo tipo si può aggiungere `[]` alla fine della stringa `ClassName`), e `ObjId` può essere utilizzato per referenziare tale oggetto; come effetto collaterale, l'oggetto Java specificato viene creato e il suo riferimento è unificato con `ObjId`. `ObjId` può essere sia una variabile, che un atomo; trattandosi tuttavia di un riferimento ad un oggetto Java, può essere utilizzato solo nel contesto dei predicati della `JavaLibrary`. Il legame tra `ObjId` e oggetto Java dura per tutto il tempo della dimostrazione, anche in caso di backtracking.

`java_object_bt/3`

`java_object_bt(ClassName, ArgList, ObjId)` ha lo stesso comportamento di `java_object/3`, ma il legame tra il termine `ObjId` e l'oggetto Java referenziato è distrutto con il backtracking.

destroy_object/1

`destroy_object(ObjId)` è vero e come effetto collaterale distrugge, se esiste, il legame tra `ObjId` e l'oggetto Java che referenzia.

java_class/4

`java_class(ClassSource, FullClassName, ClassPathList, ObjId)` è vero se e solo `ClassSource` è una stringa che descrive il codice sorgente di una classe Java valida, il cui nome completo è `FullClassName`. `ClassPathList` è una lista (anche vuota) di percorsi che può essere richiesta per la compilazione dinamica della classe, mentre `ObjId` può essere utilizzato per referenziare un oggetto appartenente alla classe `java.lang.class` che rappresenta tale classe. Come effetto collaterale, la classe descritta (e possibilmente creata) viene caricata e resa disponibile.

java_call/3

`java_call(ObjId, MethodInfo, ObjIdResult)` è vero se e solo se `ObjId` è un termine "ground" che correntemente referenzia un oggetto Java e tale oggetto fornisce un metodo il cui nome è il funtore del termine `MethodInfo` e i cui argomenti sono gli eventuali argomenti di `MethodInfo` (che quindi può essere un termine composto); `ObjIdResult` può essere utilizzato per referenziare l'eventuale valore di ritorno del metodo. I metodi statici possono essere invocati utilizzando il termine composto `class(`*`ClassName`*`)` al posto di `ObjId` (es. `class('Math')`. Come effetto collaterale, il metodo è chiamato sull'oggetto `ObjId` e l'eventuale valore di ritorno del metodo è referenziato dal termine `ObjIdResult`. La variabile anonima usata come argomento nel termine composto `MethodInfo` è interpretata come il valore `null` di Java.

'<-'/2

`'<-'(ObjId, MethodInfo)` è vero se e solo se `ObjId` è un termine "ground" che correntemente referenzia un oggetto Java e tale oggetto

fornisce un metodo il cui nome è il funtore del termine `MethodInfo` e i cui argomenti sono gli eventuali argomenti di `MethodInfo` (che quindi può essere un termine composto). I metodi statici possono essere invocati utilizzando il termine composto `class(`*`ClassName`*`)` al posto di `ObjId` (es. `class('Math')`. Come effetto collaterale, il metodo è chiamato sull'oggetto `ObjId`. La variabile anonima usata come argomento nel termine composto `MethodInfo` è interpretata come il valore `null` di Java. È anche un operatore.

`returns/2`

`returns('<-'(ObjId, MethodInfo), ObjIdResult)` è vero se e solo se `ObjId` è un termine "ground" che correntemente referenzia un oggetto Java e tale oggetto fornisce un metodo il cui nome è il funtore del termine `MethodInfo` e i cui argomenti sono gli eventuali argomenti di `MethodInfo` (che quindi può essere un termine composto); `ObjIdResult` può essere utilizzato per referenziare l'eventuale valore di ritorno del metodo. I metodi statici possono essere invocati utilizzando il termine composto `class(`*`ClassName`*`)` al posto di `ObjId` (es. `class('Math')`. Come effetto collaterale, il metodo è chiamato sull'oggetto `ObjId` e l'eventuale valore di ritorno del metodo è referenziato dal termine `ObjIdResult`. La variabile anonima usata come argomento nel termine composto `MethodInfo` è interpretata come il valore `null` di Java. È anche un operatore.

`java_array_set/3`

`java_array_set(ObjArrayId, Index, ObjId)` è vero se e solo se `ObjArrayId` è un termine "ground" che correntemente referenzia un array Java, `Index` è un indice valido per tale array e `ObjId` è un termine "ground" che correntemente referenzia un oggetto Java che può essere inserito come elemento dell'array, in accordo al suo tipo di dato. Come effetto collaterale, `ObjId` è inserito nell'array referenziato da `ObjArrayId` nella posizione specificata da `Index` (il conteggio parte da

0, secondo la convenzione Java). La variabile anonima usata come `ObjId` è interpretata come il valore `null` di Java.

`java_array_get/3`

`java_array_get(ObjArrayId, Index, ObjIdResult)` è vero se e solo se `ObjArrayId` è un termine "ground" che correntemente referenzia un array Java, `Index` è un indice valido per tale array e `ObjIdResult` può essere utilizzato per referenziare un oggetto Java contenuto nell'array. Come effetto collaterale, `ObjIdResult` è unificato con l'oggetto in posizione `Index` nell'array referenziato da `ObjArrayId`.

`java_array_length/2`

`java_array_length(ObjArrayId, ArrayLength)` è vero se e solo se `ArrayLength` è la lunghezza dell'array referenziato dal termine `ObjArrayId`.

`java_object_string/2`

`java_object_string(ObjId, PrologString)` è vero se e solo se `ObjId` è un termine che referenzia un oggetto Java e `PrologString` è la rappresentazione dell'oggetto sotto forma di stringa (secondo la semantica dettata dal metodo `toString()` dell'oggetto Java referenziato da `ObjId`).

7.3 Gli operatori della `JavaLibrary`

`'<-'`

L'operatore infisso `'<-'` è anche un predicato. Per il suo utilizzo si veda il paragrafo precedente.

`'returns'`

L'operatore infisso `'returns'` è anche un predicato. Per il suo utilizzo si veda il paragrafo precedente.

`'.'`

L'operatore infisso `'.'` è utilizzato insieme agli pseudo metodi `set` e `get`

per accedere ai campi pubblici di un oggetto Java. La sintassi è la seguente:

ObjectRef `.` *Field* `<-` `set(`*GroundTerm*`)`

ObjectRef `.` *Field* `<-` `get(`*Term*`)`

ObjectRef è sempre il termine Prolog che identifica un oggetto Java. Il primo costrutto setta il campo pubblico *Field* al valore specificato da *GroundTerm*, che può essere un valore di un tipo di dato primitivo oppure il riferimento a un oggetto esistente: se *GroundTerm* non è un termine "ground", l'operazione di `set` fallisce. Il secondo costrutto recupera invece il valore del campo pubblico *Field* e lo unifica con *Term*, che è un termine che ha quindi la stessa semantica di *ObjectRef* (è un riferimento a un oggetto Java). I campi statici di una classe possono essere acceduti utilizzando il termine composto `class(`*ClassName*`)` al posto di *ObjectRef*. Per accedere invece agli elementi di un array bisogna invece utilizzare i predicati `java_array_set` e `java_array_get`, come descritto nel paragrafo precedente.

`'as'`

L'operatore infisso `'as'` è utilizzato per specificare esplicitamente (operazione di cast) il tipo di un oggetto:

ObjectRef `as` *ClassName*

Scrivendo così, l'oggetto referenziato da *ObjectRef* è considerato appartenere alla classe *ClassName*; *ObjectRef* e *ClassName* hanno sempre la solita semantica. L'operatore funziona anche con i tipi primitivi di Java (ad esempio è possibile scrivere `myNumber as int`), e ha l'obiettivo di richiamare i metodi con il tipo di dato esatto richiesto, oltre che di risolvere a priori possibili conflitti dovuti a overloading.

7.4 Esempi di utilizzo della `JavaLibrary`

Per illustrare meglio l'utilizzo della `JavaLibrary`, consideriamo la definizione della classe `Counter` (Figura 6).

```
public class Counter {

     public String name;

     private long value = 0;


     public Counter() {}

     public Counter(String aName) { name = aName; }


     public void setValue(long val) { value = val; }

     public long getValue() { return value; }

     public void inc() { value++; }

}
```

Figura 6: Una semplice classe Java (un contatore) usata per spiegare il funzionamento della `JavaLibrary` [1]

Nell'esempio di Figura 7 viene creato un oggetto di classe `Counter` specificando il nome `MyCounter` come argomento del costruttore; il riferimento al nuovo oggetto è legato all'atomo `myCounter`. Questo riferimento viene utilizzato per l'invocazione dei metodi dell'oggetto attraverso l'operatore `<-`, chiamando prima il metodo `setValue(5)` che è `void` e quindi non ritorna niente, poi il metodo `inc` (non vengono specificati argomenti) e infine il metodo `getValue`. Dato che `getValue` ritorna un valore intero, viene utilizzato l'operatore `returns` per ricavarne il risultato (5) e legarlo alla variabile `Value`, che viene stampata mediante il predicato `write/1`. Se `Value` è già legata a 5 prima della chiamata del metodo il predicato ha comunque successo, altrimenti fallisce. Quindi viene recuperato il nome del contatore attraverso lo pseudo metodo `get`, che lo lega alla variabile `Name` e lo stampa usando il metodo `println` fornito dal

campo statico `out` della classe `java.lang.System`. Quindi viene acceduto il campo pubblico `name` dell'oggetto `myCounter`, settando il valore `MyCounter2` come nome. Infine viene creato un array di 10 contatori, e l'oggetto `myCounter` è assegnato al suo primo elemento.

```
?-  java_object('Counter', ['MyCounter'], myCounter),
    myCounter <- setValue(5),
    myCounter <- inc,
    myCounter <- getValue returns Value,
    write(Value),

    myCounter.name <- get(Name),
    class('java.lang.System') . out <- get(Out),
    Out <- println(Name),

    myCounter.name <- set('MyCounter2'),

    java_object('Counter[]', [10], ArrayCounters),
    java_array_set(ArrayCounters, 0, myCounter).
```

Figura 7: Esempio di utilizzo della `JavaLibrary` [1]

È da notare che per poter usufruire di tutte queste funzionalità l'unica cosa di cui si ha bisogno è la presenza del file `Counter.class` nella posizione opportuna del file system; tutto il resto del lavoro viene svolto dalla `JavaLibrary`.

L'esempio di Figura 8 mostra invece l'utilizzo delle API del package Swing all'interno di un programma tuProlog: viene istanziata una finestra

di dialogo `JFileChooser` e legata alla variabile Prolog `Dialog`, che viene poi utilizzata per invocare i metodi `showOpenDialog` e `getSelectedFile` appartenenti all'interfaccia della classe `JFileChooser`.

```
Test_open_file_dialog(FileName) :-

    java_object('javax.swing.JFileChooser'',        [],
    Dialog),

    Dialog <- showOpenDialog(_),

    Dialog <- getSelectedFile returns File,

    File <- getName returns FileName.
```

Figura 8: Utilizzo di un componente Swing all'interno di un programma tuProlog. Si noti l'utilizzo della variabile anonima _ per rappresentare il valore `null` di Java [1]

L'ultimo esempio (Figura 9) mostra l'utilizzo del predicato `java_class` e quindi la possibilità di eseguire la *compilazione dinamica* di una classe.

```
?-  Source = 'public class Counter { ... }',

    java_class(Source, 'Counter', [], counterClass),

    counterClass <- newInstance returns myCounter,

    myCounter <- setValue(5),

    myCounter <- getValue returns X,

    write(X).
```

Figura 9: Il predicato `java_class` esegue la compilazione dinamica di codice Java da un programma tuProlog [1]

Il predicato `java_class` crea un'istanza di `Class` che descrive la classe pubblica dichiarata nel codice sorgente passato come primo argomento del predicato. Tale istanza, referenziata dal termine Prolog `counterClass`, può essere ad esempio utilizzata per creare istanze della classe `Counter` (attraverso il metodo `newIstance`) oppure per ispezionarne costruttori, metodi e campi attraverso la riflessione. Oltre alla stringa che descrive il codice sorgente della classe da creare/caricare e al riferimento all'istanza di `Class`, gli altri argomenti del predicato sono il nome completo della classe (per individuarla nella gerarchia dei package) e la lista di percorsi eventualmente necessari per una corretta compilazione dinamica della classe.

7.5 Eccezioni Java nella `JavaLibrary`

In tutti i predicati della `JavaLibrary` precedentemente descritti 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 anche informazioni relative all'errore che si è verificato. Serve quindi una coppia di predicati, in cui uno permetta di lanciare un'eccezione, l'altro di gestirla. Il principio di funzionamento è lo stesso dei due predicati standard `throw/1` e `catch/3`; tuttavia, come già discusso nel capitolo 4, per avere una semantica più vicina a quella del mondo Java si è preferito avere un diverso set di predicati per lanciare e gestire le eccezioni che si verificano all'interno di programmi Prolog che utilizzano oggetti Java durante la computazione. Questi predicati sono `java_throw/1` e `java_catch/3`.

7.5.1 `java_throw_1`

Il predicato `java_throw/1` ha la forma:

```
java_throw(java_exception(Cause, Message, StackTrace))
```

dove l'atomo *java_exception* prende il nome della classe della specifica eccezione Java da lanciare espressa sotto forma di stringa (ad esempio 'java.io.FileNotFoundException'), mentre i tre parametri rappresentano le tre parti che caratterizzano la tipica eccezione Java:

- 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 java_throw(Exception) lancia l'eccezione Exception che viene catturata dal più vicino antenato java_catch/3 nell'albero di risoluzione il cui secondo argomento unifica con Exception. Se non viene trovata nessuna clausola java_catch/3 che unifica, l'esecuzione fallisce. Serve per segnalare un'eccezione che si verifica all'interno di un predicato della JavaLibrary.

7.5.2 java_catch_3

Il predicato java_catch/3 ha invece la forma:

```
java_catch(JavaGoal, [(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 Java. Il terzo argomento ha invece la semantica del finally di Java e rappresenta il predicato da eseguire alla fine di Goal o di uno dei gestori. Più in particolare:

- JavaGoal deve essere un predicato della JavaLibrary
- i catcher devono avere la forma java_exception(Cause, Message, StackTrace)

- nel caso il `finally` non fosse necessario si deve passare come terzo argomento l'intero `0`

Il predicato mette in esecuzione `JavaGoal` e successivamente `Finally` se non vengono lanciate eccezioni durante l'esecuzione di `JavaGoal`. Se invece durante tale esecuzione viene lanciata un'eccezione attraverso `java_throw/1`, il sistema provvede a tagliare tutti i punti di scelta generati da `JavaGoal` (in caso di predicato non-deterministico come ad esempio `java_object_bt/3`) e prova ad unificare uno dei catcher con l'argomento di `java_throw/1`. Se l'unificazione ha successo, esegue il gestore corrispondente al catcher che unifica con l'eccezione lanciata, mantenendo le sostituzioni effettuate nell'unificazione. Se invece fallisce, il sistema continuerà a risalire l'albero di risoluzione in cerca di una clausola `java_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 `java_catch/3`. In caso di eccezioni gli effetti collaterali che si sono eventualmente verificati durante l'esecuzione di `JavaGoal` non vengono comunque distrutti.

Pertanto `java_catch/3` è vero se:

iii. `JavaGoal` e `Finally` sono veri, oppure

iv. `call(JavaGoal)` è interrotto da una chiamata a `java_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 `JavaGoal` è un predicato non-deterministico (es. `java_object_bt/3`), allora attraverso il backtracking può capitare che venga rieseguito; tuttavia in caso di eccezione viene eseguito un gestore e in seguito tutti i punti di scelta generati da `JavaGoal` sono distrutti (quindi quel gestore o un altro gestore della lista non verranno più eseguiti)

- l'esecuzione di `JavaGoal` è protetta da `java_catch/3`, mentre i gestori e `Finally` non sono protetti

7.5.3 Esempio

Vediamo ora un semplice esempio di funzionamento dei due predicati `java_throw/1` e `java_catch/3`.

Il predicato `java_object(ClassName, ArgList, ObjId)` lancia una `'java.lang.ClassNotFoundException'(Cause, Message, StackTrace)` se `ClassName` non identifica una classe Java valida. Se ad esempio la classe `Counter` è disponibile nel file system locale, l'esecuzione di:

```
java_catch(java_object('Counter',['MyCounter'],c),
[('java.lang.ClassNotFoundException'(Cause, Message,
StackTrace), write(Message))], write(+++)).
```

crea un oggetto `Counter` di nome `'MyCounter'` il cui riferimento è legato all'atomo `c`; inoltre produce l'output:

```
+++
```

perché il `finally` viene sempre eseguito. Se invece la classe `Counter` non è disponibile, non ci sono effetti collaterali è l'output è:

```
Counter+++
```

in quanto il messaggio relativo all'eccezione è la stringa `'Counter'` e poi viene eseguito il `finally`.

7.6 Modifiche apportate al motore

Sono ora analizzate le ulteriori modifiche apportate al motore per renderlo capace di gestire questa nuova tipologia di eccezioni.

7.6.1 La classe `JavaException`

La classe `JavaException` è una classe molto semplice che estende `Throwable` e rappresenta un errore che si è verificato durante l'esecuzione di un predicato della `JavaLibrary`. Una opportuna istanza di

`JavaException` viene lanciata ogni volta che un predicato della `JavaLibrary` rileva un errore durante la sua esecuzione, e il suo stato consiste nel nome della classe dell'eccezione Java corrispondente all'errore e nei suoi tre parametri che la caratterizzano (causa, messaggio associato e stacktrace). In pratica ha la stessa funzione della classe `PrologError`, ma è utilizzata dai predicati della `JavaLibrary` invece che da quelli di tutte le altre librerie di tuProlog.

7.6.2 Modifiche alla macchina a stati finiti

La Figura 10 mostra ancora una volta la nuova organizzazione della macchina a stati finiti alla base di tuProlog.

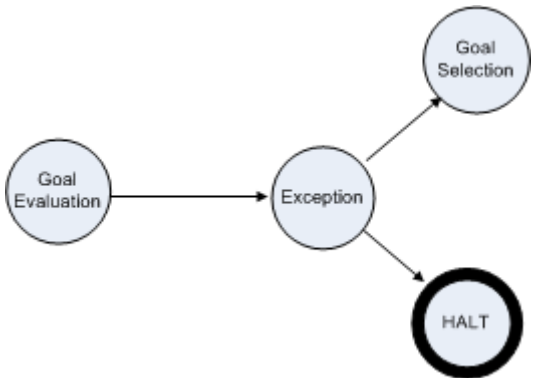

Figura 10: L'estensione della macchina a stati finiti per supportare il meccanismo della gestione degli errori [2]

Lo stato *GoalEvaluation* ha l'obiettivo di valutare un singolo subgoal. Se il funtore principale di tale subgoal rappresenta un predicato primitivo, la macchina valuta il predicato. Come ampiamente descritto nella sezione 5.3, tale valutazione si trova all'interno di un `try/catch` nel metodo `doJob()` della classe `StateGoalEvaluation`; il blocco `catch` cattura un'istanza di `Throwable` e serve appositamente per gestire le eccezioni eventualmente lanciate dal predicato. Pertanto in caso di eccezione:

i. si verifica il tipo di eccezione e si effettua un opportuno cast dell'istanza di `Throwable` in un'istanza di `PrologError` o di `JavaException` (infatti derivano entrambe da `Throwable`)

ii. il subgoal in cui si è verificato l'errore viene sostituito dal subgoal `throw(Error)` o `java_throw(Exception)`, in base al tipo di eccezione che è stata lanciata. La macchina ricava le informazioni sull'errore dall'istanza di `PrologError` o di `JavaException` catturata, e assegna il termine `throw/1` o `java_throw/1` alla variabile `currentGoal` del corrente `ExecutionContext` del motore

iii. infine in entrambi i casi il motore transita nello stato *Exception*. Infatti, essendo lo stato *GoalEvaluation* unico sia per i predicati della `JavaLibrary` che per tutti gli altri, si è scelto di avere anche un unico stato di *Exception* per entrambe le tipologie di errore

La classe `StateException` modella invece il nuovo stato *Exception* che ha il compito di gestire le eccezioni. Come per gli altri stati, il comportamento è definito nel metodo `doJob()` della classe. La macchina come prima cosa controlla la variabile `currentGoal` del corrente `ExecutionContext` del motore, appena settata nello stato *GoalEvaluation*. Infatti se si tratta di un subgoal `throw/1` allora il motore dovrà effettuare una visita all'indietro dell'albero di risoluzione (composto di oggetti `ExecutionContext`) per ricercare un subgoal `catch/3` il cui secondo argomento è unificabile con l'argomento dell'eccezione lanciata, altrimenti si tratta di un subgoal `java_throw/1` e pertanto il motore dovrà ricercare un subgoal `java_catch/3`. La gestione dei due tipi di errore avviene quindi nello stesso stato (e quindi nella stessa classe), ma comunque in modo differenziato; infatti le operazioni da svolgere nei due casi sono leggermente diverse e verranno quindi invocati due metodi privati differenti in base al tipo di errore.

Durante la ricerca di un subgoal `java_catch/3` che abbia un catcher unificabile con l'argomento dell'eccezione lanciata, ogni `ExecutionContext` dell'albero di risoluzione 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 `java_catch/3`, la macchina:

i. taglia tutti i punti di scelta generati da `Goal` (attraverso il metodo `cut()` della classe `EngineManager`)

ii. unifica l'argomento di `java_throw/1` con il giusto catcher (attraverso il metodo `unify()` della classe `Term`)

iii. inserisce il gestore dell'errore e l'eventuale `finally` in testa alla lista dei subgoal da eseguire, mantenendo le sostituzioni effettuate durante il processo di unificazione per quel che riguarda il gestore (viene utilizzato il metodo `pushSubGoal()` della classe `EngineManager`)

Infine la macchina transita nello stato *GoalSelection* se è stato trovato un nodo `java_catch/3` appropriato e quindi l'errore può essere gestito, oppure nello stato *HALT* se tale nodo non è stato trovato: l'errore pertanto non può essere gestito e la macchina termina l'esecuzione.

7.6.3 Implementazione di `java_throw/1` e `java_catch/3`

L'implementazione del predicato `java_catch(JavaGoal, [(Catcher1, Handler1), …, (CatcherN, HandlerN)], Finally)` è molto semplice, 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 particolare dallo stato *Exception*. È implementato in Prolog perché `JavaGoal` potrebbe essere un predicato non-deterministico (es. `java_object_bt/3`) e in tal caso deve essere rieseguito attraverso il meccanismo del backtracking. Il predicato non fa altro che chiamare `JavaGoal` e `Finally`, poi 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 `JavaLibrary`, tale libreria è la sua collocazione più ovvia.

Il predicato `java_throw(Exception)` invece non deve neanche 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*. L'effettiva eccezione è costituita dall'istanza di `JavaException` lanciata dai predicati della `JavaLibrary` e catturata nello stato *GoalEvaluation*.

7.6.4 Modifiche ai predicati della `JavaLibrary`

Così come è stato fatto per le altre librerie, anche i predicati della `JavaLibrary` devono essere modificati in modo che essi non falliscano in caso di errori, ma lancino le opportune eccezioni. I vari predicati lanceranno pertanto un'istanza di `JavaException` corrispondente all'eventuale errore: tale istanza sarà catturata nello stato *GoalEvaluation* del motore, mentre le informazioni sull'errore che si è verificato saranno poi utilizzate dallo stato *Exception* per ricercare un subgoal `java_catch/3` opportuno che sia in grado di gestirlo.

La `JavaLibrary` presenta predicati implementati sia in Java 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 `JavaException`) vengono lanciate direttamente dai corrispondenti metodi Java ogniqualvolta si verifica un errore, mentre nel secondo caso sono lanciate da metodi "guardia" (sempre espressi in Java) invocati per controllare i parametri prima dell'esecuzione del predicato Prolog. Le eccezioni inoltre possono derivare sia dall'invocazione di un predicato della `JavaLibrary` con parametri errati, sia dalla chiamata di un metodo di qualsiasi classe Java esterna al motore. In ogni caso vale ancora 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. Infine un utente può anche decidere di non gestire le eccezioni lanciate dai predicati della `JavaLibrary`: in questo caso l'esecuzione fallirà, e l'estensione è come se non ci fosse.

7.7 Eccezioni lanciate dai predicati

Nel seguito verranno descritte le eccezioni lanciate dai i predicati della `JavaLibrary` e in quali circostanze ciò può avvenire. Si noti che i

predicati possono lanciare innumerevoli tipi di eccezioni, visto che sono utilizzati per invocare classi Java 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).

`java_object(ClassName, ArgList, ObjId)`

- `'java.lang.ClassNotFoundException'(Cause, Message, StackTrace)`

  se `ClassName` non identifica una classe Java valida

- `'java.lang.NoSuchMethodException'(Cause, Message, StackTrace)`

  se il costruttore specificato non è stato trovato

- `'java.lang.reflect.InvocationTargetException'(Cause, Message, StackTrace)`

  se gli argomenti del costruttore non sono validi (ad esempio non sono "ground")

- `'java.lang.Exception'(Cause, Message, StackTrace)`

  se `ObjId` già riferisce un altro oggetto nel sistema

`java_object_bt(ClassName, ArgList, ObjId)`

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

`java_class(ClassSourceText, FullClassName, ClassPathList, ObjId)`

- `'java.lang.ClassNotFoundException'(Cause, Message, StackTrace)`

  se la classe non può essere localizzata nella gerarchia dei package così come specificato

- `'java.io.IOException'(Cause, Message, StackTrace)`

  se `ClassSourceText` contiene errori

- `'java.lang.Exception'(Cause, Message, StackTrace)`

  se `ObjId` già riferisce un altro oggetto nel sistema

`java_call(ObjId, MethodInfo, ObjIdResult)`

- `'java.lang.NoSuchMethodException'(Cause, Message, StackTrace)`

  se si invoca un metodo non valido per l'oggetto o per la classe oppure se gli argomenti del metodo non sono validi

`'<-'(ObjId, MethodInfo)`

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

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

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

`java_array_set(ObjArrayId, Index, ObjId)`

- `'java.lang.IllegalArgumentException'(Cause, Message, StackTrace)`

  se `ObjArrayId` non riferisce alcun oggetto del sistema, se `Index` non è un indice corretto oppure se `ObjId` non è un valore corretto per l'array

`java_array_get(ObjArrayId, Index, ObjIdResult)`

- `'java.lang.IllegalArgumentException'(Cause, Message, StackTrace)`

  se `ObjArrayId` non riferisce alcun oggetto del sistema oppure se `Index` non è un indice corretto

7.8 Collaudo

Il collaudo del funzionamento congiunto dei due nuovi predicati `java_throw/1` e `java_catch/3` ha l'obiettivo di verificare il rispetto dei seguenti requisiti (per ognuno di essi è riportato anche come la verifica è stata eseguita; si suppone che la classe `Counter` non sia disponibile e che quindi i tentativi di istanziazione di oggetti di tale classe provochino errori):

i. il gestore deve essere eseguito con le sostituzioni effettuate nel processo di unificazione tra l'eccezione e il catcher; successivamente deve essere eseguito il `finally`

```
?- java_catch(java_object('Counter',
['MyCounter'], c),
[('java.lang.ClassNotFoundException'(Cause,
Message, _), X is 2+3)], Y is 2+5).

yes.

Cause/0

Message/'Counter'

X/5

Y/7
```

ii. deve essere eseguito il più vicino antenato `java_catch/3` nell'albero di risoluzione avente un catcher unificabile con l'eccezione lanciata

```
?- java_catch(java_object('Counter',
['MyCounter'], c),
[('java.lang.ClassNotFoundException'(Cause,
Message, _), true)], true),
java_catch(java_object('Counter', ['MyCounter2'],
c2), [('java.lang.ClassNotFoundException'(C, M,
```

```
_), X is 2+3)], true)..

yes.

Cause/0

Message/'Counter'

C/0

M/'Counter'

X/5
```

iii. l'esecuzione deve fallire se si verifica un errore durante l'esecuzione di un goal e non viene trovato nessun nodo `java_catch/3` avente un catcher unificabile con l'argomento dell'eccezione lanciata

```
?- java_catch(java_object('Counter',
['MyCounter'], c), [('java.lang.Exception'(Cause,
Message, _), true)], true).

no.
```

iv. `java_catch/3` deve fallire se il gestore è falso

```
?- java_catch(java_object('Counter',
['MyCounter'], c), [('java.lang.Exception'(Cause,
Message, _), false)], true).

no.
```

v. il `finally` deve essere in caso di successo del goal passato come primo argomento

```
?- java_catch(java_object('java.util.ArrayList',
[], l), [E, true], X is 2+3).
```

```
yes.

X/5
```

vi. `java_catch/3` deve fallire se si verifica un'eccezione durante l'esecuzione del gestore

```
?- java_catch(java_object('Counter',
['MyCounter'], c),
[('java.lang.ClassNotFoundException'(Cause,
Message, _), java_object('Counter',
['MyCounter'], c))], true).

no.
```

vii. deve essere eseguito il gestore corretto tra quelli presenti nella lista passata come secondo argomento

```
?- java_catch(java_object('Counter',
['MyCounter'], c), [('java.lang.Exception'(Cause,
Message, _), X is 2+3),
('java.lang.ClassNotFoundException'(Cause,
Message, _), Y is 3+5)], true).

yes.

Cause/0

Message/'Counter'

Y/8
```

La verifica ha evidenziato il rispetto di tutti i requisiti di funzionamento sopra citati.

Successivamente si è collaudato il lancio delle principali eccezioni (come detto in precedenza, non è possibile prevederle tutte a priori) che i predicati

della `JavaLibrary` sono ora in grado di generare, provando ad invocare un predicato con dei parametri errati e verificando che il motore lanci l'eccezione corretta. Ad esempio, per verificare che `java_object(ClassName, ArgList, ObjId)` lancia un'eccezione se `ClassName` non identifica una classe Java valida:

i. si è eseguito il predicato all'interno di un blocco `java_catch/3`

ii. si è verificato che la sintassi dell'eccezione lanciata dal predicato è corretta in base all'invocazione. Nell'esempio l'eccezione che si deve generare è:

```
'java.lang.ClassNotFoundException'(Cause,
Message, StackTrace)
```

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

Infine, come collaudo generale per tutta l'estensione, si è verificato (con successo) che tutti i test effettuati in precedenza (descritti nei precedenti capitoli) continuassero ad avere esito positivo.

Conclusioni

Lo scopo della tesi era estendere l'interprete tuProlog al fine di supportare il meccanismo linguistico che Prolog offre per la gestione delle eccezioni, specificato nello standard ISO/IEC 13211-1.

Per raggiungere questo obiettivo è stato necessario studiare a fondo l'architettura del motore, in modo da implementare il meccanismo tenendo conto delle peculiarità tipiche di tuProlog. Il supporto implementato risulta compatibile sia con lo standard, che con le altre implementazioni di riferimento (SICStus Prolog, SWI-Prolog, ecc…). Oltre all'introduzione delle eccezioni nei predicati Prolog di libreria, le eccezioni sono supportate anche per la parte del motore relativa all'utilizzo di oggetti e classi Java all'interno di programmi Prolog: le eccezioni lanciate dagli oggetti Java utilizzati durante la computazione non vengono più catturate dai predicati della `JavaLibrary` come avveniva in precedenza (per poi far fallire il predicato), ma vengono inoltrate.

In entrambi i casi la gestione delle eccezioni è comunque a discrezione dell'utente, che può infatti anche decidere di non gestirle; in questo caso il motore si comporta come se le eccezioni non ci fossero (il predicato fallisce). Pertanto l'introduzione di tale meccanismo offre una possibilità in più all'utente, ma i programmi scritti in precedenza (e che quindi non gestiscono le eccezioni) continuano a funzionare. Questo risultato è confermato anche dal fatto che tutte le suite di test esistenti hanno successo anche con il nuovo motore.

Oltre ad aver verificato ciò, il collaudo dell'estensione si è svolto seguendo due direttive: l'aderenza dei due predicati `throw/1` e `catch/3` allo standard e la verifica che i predicati di libreria lancino le opportune eccezioni. Tutti i numerosi test svolti hanno dato esito positivo.

Infine la tesi contiene anche una parte manualistica in cui vengono descritte tutte le eccezioni lanciate da tutti i predicati delle librerie di tuProlog, e in quali circostanze ciò può avvenire.

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] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, G. Puebla. *The CIAO Prolog System*. April 2006. http://www.ciaohome.org/.

[5] Mats Carlsson et al. *SICStus Prolog User's Manual*. September 2009. http://www.sics.se/sicstus/.

[6] Jan Wielemaker. *SWI-Prolog 5.6 Reference Manual*. August 2008. http://www.swi-prolog.org/.

[7] jTrolog – Java Throndeim Prolog. https://jtrolog.dev.java.net/.