

ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DEIS

TESI DI LAUREA

In

Linguaggi e Modelli Computazionali LS

Integrazione di codice tuProlog in linguaggi .NET: approcci a confronto

CANDIDATO

Fabio Gravina

RELATORE

Chiar.mo Prof. Enrico Denti

Anno Accademico 2011/2012

Sessione II

Indice

Introduzione	5
1. tuProlog.....	7
1.1 Caratteristiche.....	7
1.2 Architettura.....	8
1.2.1 Gestione del Motore Inferenziale	8
1.2.2 Gestione delle Teorie	8
1.2.3 Gestione dei Predicati.....	9
1.2.4 Gestione delle Librerie	9
1.3 Motore Inferenziale	10
1.4 Interoperabilità	12
2. Ambiente di sviluppo .Net	15
2.1 .Net Framework.....	15
2.2 Visual Studio	16
2.3 Estensioni per Visual Studio	16
2.3.1 Creare l'involucro di una nuova estensione	17
2.3.2 Analisi del Manifest File	17
2.4 Template	18
2.4.1 Creare un nuovo Template Item	19
2.5 Generatore di Codice.....	20
3. Programmazione multi-paradigma in Prolog e .Net	22
3.1 tuProlog.Net	22
3.2 IKVM	23
3.3 Approcci alla programmazione multi-paradigma	23
3.3.1 Usare .Net da tuProlog: OOLibrary	24
3.3.2 Usare tuProlog da .Net: tuProlog API	25

3.3.3 Accrescere tuProlog via .Net: nuove librerie.....	25
3.3.4 Accrescere .Net via tuProlog: P@J framework.....	26
4. Approccio basato sull'utilizzo di P@J su .Net.....	27
4.1 Stato Attuale	27
4.2 Tipi Generici.....	27
4.3 Tipi Generici in Java e .Net.....	29
4.3.1 Implementazione dei Tipi Generici in Java.....	29
4.3.2 Implementazione dei Tipi Generici in .Net	30
4.4 Considerazioni su P@J in .Net.....	31
4.4.1 P@J e i Generici.....	31
5. Approccio basato sui generatori di codice per .Net	34
5.1 Requisiti	34
5.2 Analisi dei Requisiti	34
5.2.1 Modello della classe generata.....	35
5.2.2 Personalizzazione della classe generata	36
5.2.3 Modalità di generazione	37
5.3 Progetto	39
5.3.1 Vincoli Progettuali.....	39
5.3.2 Diagramma della classi	39
5.4 Implementazione di PrologVSExtension	42
5.4.1 PrologItemTemplate	43
5.4.2 PrologGenerator	45
6. Collaudo della tecnologia	50
6.1 Sezione Logica	51
6.1.1 Permutazioni.....	51
6.1.2 Derivate	52

6.2 Interfaccia Grafica.....	54
6.3 Osservazioni.....	57
7. Conclusioni	58
Bibliografia	60

Introduzione

TuProlog^[1] è un interprete Prolog. Interamente scritto in Java^[2], minimale, dinamicamente configurabile ed estensibile tramite librerie, progettato in modo da supportare pienamente la programmazione multi-paradigma tramite una piena integrazione fra il paradigma logico dato dal linguaggio Prolog e il paradigma a oggetti, rappresentato primariamente da Java – ma anche, nella versione .NET, dai linguaggi disponibili su tale piattaforma.

TuProlog supporta la programmazione multi-paradigma tra Prolog e Java in quattro dimensioni, basate sull'uso e l'arricchimento reciproco dei due linguaggi.

E' possibile da un lato utilizzare Java da Prolog, (mediante la JavaLibrary) o Prolog da Java (utilizzando le Java API), ma anche usare un linguaggio come mezzo per “arricchire” l'altro, implementando in un linguaggio una estensione per l'altro: in particolare tuProlog permette di arricchire il Prolog via Java sviluppando in Java nuove librerie per tuProlog, ma anche, dualmente, di arricchire Java permettendo di scrivere in Prolog l'implementazione di metodi Java, mediante il framework P@J incluso nel sistema.

Quest'ultimo approccio è basato sulle Java Annotation: in quanto il codice tuProlog che implementa un metodo Java è specificato sotto forma di annotazione nel sorgente java.

Pur essendo scritto primariamente in Java, tuProlog è un prodotto multi-piattaforma: la versione .Net in particolare è ottenuta mediante traduzione automatica del bytecode Java in codice .NET, tramite lo strumento IKVM. Sebbene questo processo traduca tutti i componenti tuProlog, compreso – quindi- il framework P@J, l'utilizzo di tale componente in ambito .NET risulta attualmente limitato alle sole applicazioni scritte originariamente in Java (ovvero, in cui il codice prolog viene usato per implementare metodi java poi tradotti in .NET), mentre il tentativo di utilizzarlo nativamente in applicazioni .Net (ovvero, usando Prolog per implementare un metodo C# o VB), dà luogo a eccezioni a tempo di esecuzione. Questo problema non dipende da banali bug o imperfezioni del codice,

ma da questioni più profonde legate all'interazione fra i diversi linguaggi (e con lo strumento di traduzione automatica) sia a compile time che a run time. Pertanto, scopo di questa tesi sarà l'analisi approfondita di tali problematiche di interazione, studiando le possibili soluzioni e le loro implicazioni sul processo di produzione e manutenzione delle applicazioni multi-paradigma che tuProlog mira a supportare.

In questo quadro, fra i vari approcci possibili sarà esplorato in particolare un approccio basato sul concetto di *generatore di codice*, una funzionalità supportata dagli strumenti di sviluppo del mondo .NET che ben si presta a svolgere un ruolo interessante nel contesto del problema in questione.

La tesi è dunque strutturata come segue. Nel capitolo 1 si analizzeranno le caratteristiche generali di tuProlog. Successivamente nel capitolo 2 sarà esplorato il framework .Net e il suo ambiente di sviluppo di riferimento Visual Studio. Nel capitolo 3 saranno approfonditi i diversi approcci di integrazione tra tuProlog e .Net, al fine di delineare la situazione attuale.

Il capitolo 4 sarà incentrato sullo studio della interazione tra tuProlog e .Net tramite il framework P@J allo scopo di identificare cause e possibili soluzioni dell'attuale malfunzionamento.

Il capitolo 5 affronterà lo studio approfondito del nuovo approccio basato sui generatori di codice. Di seguito verrà progettata una soluzione completa in grado di recuperare le funzionalità previste dal framework P@J. Tale soluzione sarà incentrata nell'ambito di un processo di generazione e produzione del software alternativo e originale, basato sulla generazione automatica di classi .Net che incorporino la versione (C#, VB) del codice prolog originale.

Infine nel capitolo 6 sarà illustrata l'implementazione di una applicazione che ha lo scopo di collaudare il generatore di codice realizzato.

Capitolo 1

tuProlog

1.1 Caratteristiche

TuProlog^[1] è un interprete Prolog. Interamente scritto in Java^[2], minimale, dinamicamente configurabile ed estensibile tramite librerie, esso è progettato in modo da supportare pienamente la programmazione multi-paradigma tramite una piena integrazione fra il paradigma logico dato dal linguaggio Prolog e il paradigma a oggetti, rappresentato primariamente da Java – ma anche, nella versione .NET, dai linguaggi disponibili su tale piattaforma.

Esso è distribuito in un unico archivio di ridotte dimensioni e ha come unico prerequisito la presenza della Java Virtual Machine (JVM) o del corrispondente strato Common Language Runtime.

TuProlog è stato sviluppato con un meccanismo di estensione basato sul caricamento statico o dinamico di librerie, così da poter facilmente aggiungere funzionalità non presenti nel motore.

Grazie alla libreria JavaLibrary è stato possibile integrare il tuProlog con Java. La libreria offre una integrazione bidirezionale che permette di utilizzare entità Java all'interno di tuProlog il quale li gestisce come normali termini. In questo modo è possibile estendere le funzionalità di tuProlog accedendo a tutte quelle di Java.

Allo stesso modo è possibile rappresentare, lato Java, il motore tuProlog come un semplice oggetto che sarà utilizzato secondo il paradigma Object Oriented, cioè tramite l'invocazione di metodi, come ad esempio la SetTheory per caricare una base di conoscenza all'interno del motore, oppure la Solve per la risoluzione di un *goal*.

Ultima, ma non meno importante, delle caratteristiche salienti di tuProlog è l'interoperabilità, in particolare attraverso TCP/IP, RMI e CORBA.

1.2 Architettura

L'architettura di tuProlog è suddivisibile in una serie di gestori che si occupano di precise funzionalità, ma che sono nascosti al programmatore finale attraverso una visione semplificata che si concretizza nell'utilizzo di un singolo oggetto Java.

Questi gestori, che adesso andremo ad analizzare più in dettaglio, si occupano rispettivamente di:

- Gestione del motore inferenziale;
- Gestione delle teorie;
- Gestione dei predicati primitivi;
- Gestione delle librerie.

1.2.1 Gestione del Motore Inferenziale

Questo componente ha lo scopo di disaccoppiare il motore inferenziale vero e proprio, realizzato attraverso una macchina a stati finiti, e il resto dei componenti con cui il motore coopera. Potremmo perciò definirlo come una interfaccia di accesso al motore inferenziale che offre a quest'ultimo anche alcune funzionalità di supporto alla macchina a stati finiti.

1.2.2 Gestione delle Teorie

Una teoria Prolog è normalmente espressa in formato testuale e necessita di essere trasformata in una serie di oggetti, detti termini, che il motore inferenziale possa comprendere e gestire. Il compito del gestore delle teorie è proprio quello di creare la base di conoscenza utilizzando un parser per il riconoscimento delle frasi espresse in Prolog. Il gestore, nel caso in cui non ci siano errori, fornisce al motore inferenziale gli oggetti con tutte le informazioni necessarie alla loro elaborazione, oppure genera degli errori inerenti la correttezza della teoria.

1.2.3 Gestione dei Predicati

I predicati tuProlog sono gestiti in modo diverso in funzione della loro natura, infatti tuProlog ne gestisce di tre tipi: *built-in*, *teoria* e *primitivi*.

I predicati built-in sono i predicati Prolog, cioè quelli definiti nel core del motore inferenziale. I predicati di teoria sono quelli scritti dall'utente nella sua teoria. Entrambe queste tipologie sono gestite direttamente dal motore inferenziale.

I predicati primitivi, infine, sono predicati corrispondenti a metodi Java che vengono gestiti ed eseguiti come tali dal gestore quando ne riconosce uno all'interno della nostra teoria.

1.2.4 Gestione delle librerie

Il gestore delle librerie è il componente che ci permette di estendere le capacità del motore inferenziale caricando e scaricando le librerie sia in fase di configurazione, cioè prima di avviare il motore tuProlog, sia a run time, cioè mentre il motore sta eseguendo.

Le librerie tuProlog sono collezioni di predicati e funtori che permettono all'utente di aumentare le capacità del motore inferenziale, che di base è volutamente minimale, e difatti tra i predicati built-in vi sono solo semplici espressioni come: *true*, *fail*, *halt* e *call*.

Alcune librerie sono caricate di default nel motore tuProlog, tra cui:

- BasicLibrary definisce tutti i predicati base del linguaggio Prolog, fatta eccezione per quelli built-in e quelli di I/O;
- IOLibrary fornisce i predicati Prolog di I/O;
- ISOLibrary fornisce una serie di predicati standard ISO, che consentono al tuProlog di essere ISO-Compliant;
- JavaLibrary, che come già accennato nella gestione dei predicati primitivi, definisce tutti i predicati che permettono l'interazione di tuProlog con Java.

Le librerie possono essere sviluppate in tuProlog, in Java o in entrambi i linguaggi. Sviluppando in Java si potranno realizzare solo predicati deterministici, invece utilizzando il prolog si avrà la possibilità di restituire soluzioni multiple, cioè avere anche predicati non deterministici; infine si possono prevedere dei predicati tuProlog che usano funzioni Java e viceversa.

1.3 Motore inferenziale

Il cuore di tuProlog è il suo motore inferenziale, che è stato realizzato come una macchina a stati finiti, strutturata come in Figura 1-1. Questo automa si può logicamente scomporre in:

- Uno stato iniziale;
- Cinque stati intermedi in cui l'automa transita durante l'esecuzione;
- Quattro stati finali che identificano le possibili conclusioni della elaborazione del goal.

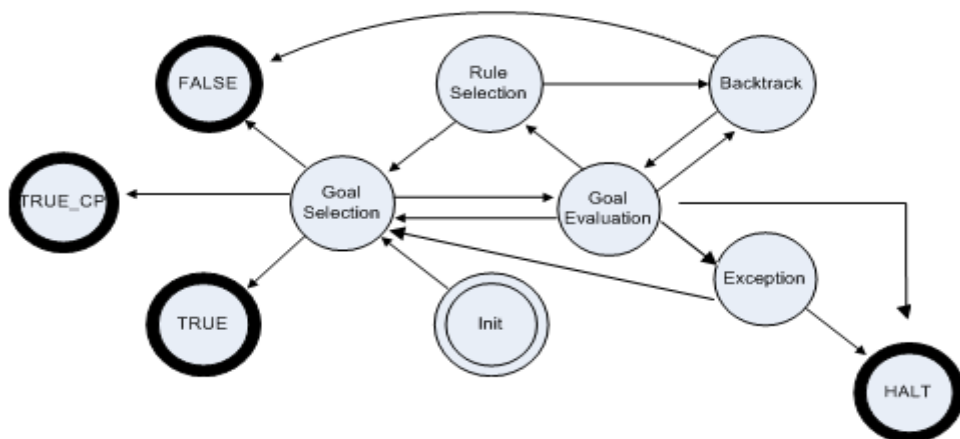


Figura **Error! No text of specified style in document..1**: TuProlog finite state machine

Punto di accesso al motore inferenziale è lo stato di *Init*, in questa fase viene inizializzato il motore tuProlog estraendo dalla query tutti i subgoal che devono

essere valutati, e per ognuno di essi viene creato un oggetto che rappresenta il contesto di esecuzione.

Lo stato successivo è *GoalSelection* nel quale viene recuperato il prossimo subgoal da valutare. A questo punto possono verificarsi quattro situazioni:

- Non esistono ulteriori subgoal e ci sono altri punti di scelta aperti. In questo caso il sistema si porta nello stato *TRUE_CP*, che permetterà di richiedere le altre soluzioni;
- Non esistono ulteriori subgoal e non ci sono altri punti di scelta aperti. In questo caso il sistema si porta nello stato *TRUE* e il processo termina;
- Esiste almeno un altro subgoal ed è un numero. In questo caso la dimostrazione fallisce e il sistema si porta in stato *FALSE*;
- Esiste almeno un altro subgoal ed è una variabile. In questo caso il sistema si predispone a continuare l'elaborazione e si porta nello stato di *GoalEvaluation*.

Lo stato *GoalEvaluation* si occupa di valutare il subgoal precedentemente estratto. Se il subgoal in questione è un predicato primitivo la sua valutazione può dare esito positivo riportando il sistema nello stato *GoalSelection* per estrarre un nuovo subgoal da valutare, o esito negativo portando il sistema nello stato *Backtrack*. Se invece il subgoal non è un predicato primitivo, è necessario andare a cercare nella base di conoscenza una clausola compatibile: a questo provvede lo stato *RuleSelection*. La valutazione del subgoal può generare in alcuni casi delle eccezioni, in tal caso il sistema si porta nello stato *Exception*; nel caso di errori, non gestiti tramite eccezioni, il sistema si porta nello stato di *HALT*.

Se lo stato di *RuleSelection* non ha trovato alcuna clausola compatibile con il subgoal allora il sistema si porta nello stato *Backtrack*. Se invece la ricerca è andata a buon fine, viene creato un nuovo contesto di esecuzione, il subgoal viene unificato con la testa della clausola e il corpo di quest'ultima viene aggiunto alla lista dei risolventi, sostituendo il subgoal appena unificato.

Lo stato di *Backtrack*, nel caso in cui non siano state trovate clausole compatibili, è utilizzato dal sistema per riportarsi allo stato precedente, ripristinando lo stato di

variabili, risolvibili e del contesto di esecuzione. Dopodiché il sistema si porta nello stato di *GoalEvaluation* che fa ripartire il processo di risoluzione.

Lo stato *Exception* si occupa della gestione delle eccezioni; quando l'automa si trova in questo stato visita all'indietro l'intero stack di esecuzione alla ricerca di un subgoal *catch/3* il cui secondo argomento unifichi con l'argomento dell'eccezione lanciata. Se non trova alcun subgoal *catch/3* compatibile, transita nello stato *HALT* e si ferma; altrimenti inserisce il gestore dell'errore in testa alla lista dei subgoal da eseguire e si riporta nello stato *GoalSelection*.

1.4 Interoperabilità

tuProlog in quanto linguaggio di logica non è stato pensato per essere utilizzato nella realizzazione di applicazioni complete di interfaccia utente, connessioni di rete o gestione di basi di dati. Per poter sfruttare le potenzialità dei linguaggi di logica molte implementazioni di Prolog mettono a disposizione alcuni meccanismi che gli permettano di dialogare con altri linguaggi di programmazione, dando vita ad un ambiente di sviluppo completo. Anche tuProlog è stato esteso per offrire questo tipo di interoperabilità, in particolare con Java e con la piattaforma .NET.

La prima, e più naturale integrazione che è stata sviluppata, è quella con il linguaggio Java con il quale tuProlog dialoga attraverso la *JavaLibrary* a cui abbiamo già accennato nel paragrafo 1.2.4. Grazie a questa libreria è possibile creare applicazioni: con interfaccia grafica swing, connesse in rete o che utilizzino database come fonte di dati. Questi dati però potranno essere elaborati da uno o più engine tuProlog.

Un diverso approccio di integrazione, atto ad accrescere le funzionalità di Java tramite tuProlog, è il framework *P@J*. Quest'ultimo è basato sulle *Java Annotation*, tramite le quali permette di specificare codice tuProlog come implementazione di metodi Java, ciò significa che ogni qual volta il metodo Java annotato verrà chiamato il sistema eseguirà il codice tuProlog specificato nella annotazione.

In Tabella 1.2 è riportato un esempio completo di utilizzo del framework P@J. In questo caso il metodo astratto `Perm.permutation` è stato decorato con l'annotazione `@PrologMethod`. L'effetto di questa annotazione è che a runtime verrà creata una implementazione del metodo `permutation` capace di risolvere il problema della permutazioni usando la teoria espressa nella annotazione.

```
//A Java class augmented via Prolog
abstract class Perm{
    @PrologMethod ( clauses = {
        "permutation([],[])." ,
        "permutation(U, [X|V]):-remove(U,X,Z),permutation(Z,V)." ,
        "remove([X|T],X,T)." ,
        "remove([X|U],E, [X|V]):-remove(U,E,V)."
    }
)
    public abstract < $X extends List<Int>, $Y extends List<Int> >
        Iterable<$Y> permutation($X list);
}

//A sample client class
public class PJexample {
    public static void main(String[] args) throws Exception {
        java.util.Collection<Integer> v = java.util.Arrays.asList(1,2,3);
        Perm p=PJ.newInstance(Perm.class);
        for (List<Int> list : p.permutation(new List<Int>(v))) {
            System.out.println(list.toJava());
        }
    }
}

// Output printed:
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Tabella 1.2: Esempio completo di utilizzo del framework P@J

L'interoperabilità con la piattaforma .Net non è implementata in modo diretto, cioè attraverso lo sviluppo di librerie ad hoc, come la JavaLibrary, o con strumenti come P@J, ma passa per un processo di traduzione del codice di tuProlog in .Net; che verrà approfondito nel capitolo 3.

Capitolo 2

Ambiente di sviluppo .Net

2.1 .Net Framework

.Net Framework^[3] è una piattaforma di sviluppo per applicazioni e servizi web basati sulla implementazione della Common Language Infrastructure (CLI^[4]) realizzata da Microsoft. La CLI è la specifica aperta, regolamenta dall'ISO, in cui sono descritte la struttura, il funzionamento e le caratteristiche base di questa piattaforma di sviluppo. Esistono altre implementazioni di questa specifica tra cui i progetti MONO^[5] e DotGNU^[6], entrambi concentrati principalmente sulla piattaforma Linux.

Seguendo le indicazioni della CLI, .Net mette a disposizione del progettista una suite completa di linguaggi in grado di essere compilati in un linguaggio intermedio chiamato Common Intermediate Language (CIL^[4]) ed eseguiti su una macchina virtuale nota come Common Language Runtime (CLR^[7]).

.Net condivide alcune caratteristiche con Java: entrambi gli ambienti sono basati sulle rispettive macchine virtuali che eseguono il codice intermedio, consentendo così la separazione tra il sistema operativo e le applicazioni, rendendo queste ultime portabili. Ciò rende quindi possibile eseguire il CIL ottenuto dalla compilazione con la piattaforma .Net su un calcolatore con in esecuzione la macchina virtuale CLR Mono, in quanto entrambe si rifanno alla stessa specifica.

CLI è concepita per essere compatibile con molti linguaggi di programmazione Object Oriented. Questo aspetto è stato fondamentale per Microsoft nello sviluppo di .Net, il quale ha goduto di ampia diffusione sia dal suo rilascio grazie al fatto che tra i linguaggi compatibili ci sono anche Visual Basic^[8] e Visual C++^[9], linguaggi di riferimento nella programmazione in ambiente Windows. Ha inoltre introdotto due nuovi linguaggi: C#^[10], derivato da Delphi^[11], C++^[12] e Java; e Visual J#^[13], una rivisitazione di Java in chiave Microsoft.

2.2 Visual Studio

L'ambiente di sviluppo (Integrated Development Enviroment, IDE) Visual Studio^[14], sviluppato da Microsoft, fornisce l'accesso a tutti strumenti necessari per lo sviluppo di una applicazione per il framework .Net, tra cui: i compilatori, gli editor e i debugger per i diversi linguaggi della piattaforma. Visual Studio è stato progettato e sviluppato, similmente ad altri IDE (Eclipse, Netbeans), come un contenitore di funzionalità alle quali fornisce tutti i necessari meccanismi di accesso ai dati e la possibilità di scambiarsi informazioni. Questa funzionalità possono essere implementate e installate in due forme diverse: gli *add-in* e le *estensioni*.

Queste due categorie di componenti si differenziano per le API a cui hanno accesso, e quindi hanno differenti peculiarità. Gli add-in sono principalmente utilizzati per personalizzare e automatizzare l'IDE. Le estensioni, invece, sono focalizzate sulla bussiness logic, quindi hanno come target una determinata classe di progetti (C#, VB, Console, Windows Form, etc...) o classe di file (xml, Resource, etc..).

In questo modo anche componenti come i compilatori e gli editor sono visti come funzionalità aggiuntive che vengono caricate dentro l'IDE solo quando necessarie. Ciò significa che il compilatore e l'editor per codice C# saranno caricati solo se l'utente aprirà o creerà un progetto di tipo C#.

2.3 Estensioni per Visual Studio

Nel nostro contesto, non sono necessarie personalizzazioni dell'IDE ma bensì l'aggiunta di una unità di bussiness logic che operi con i file Prolog. Per questa ragione, come si vedrà nei capitoli successivi, si è optato per la realizzazione di una estensione; tale meccanismo verrà adesso brevemente approfondito.

La creazione di un'estensione per Visual Studio ha come unico pre-requisito l'installazione del Visual Studio Software Development Kit (SDK)^[15]. Questo kit viene distribuito separatamente rispetto alla installazione base di Visual Studio e fornisce anche alcuni esempi e la documentazione sul processo di sviluppo.

2.3.1 Creare l'involucro di una nuova estensione

La creazione di una nuova estensione si basa su una breve procedura guidata messa a disposizione dal Visual Studio SDK.

Le estensioni di Visual Studio sono distribuite attraverso i file VSIX (Visual Studio Installer eXtension), i quali incapsulano l'estensione stessa e alcune informazioni aggiuntive atte a verificare la compatibilità dell'estensione con la versione di Visual Studio installata nel sistema.

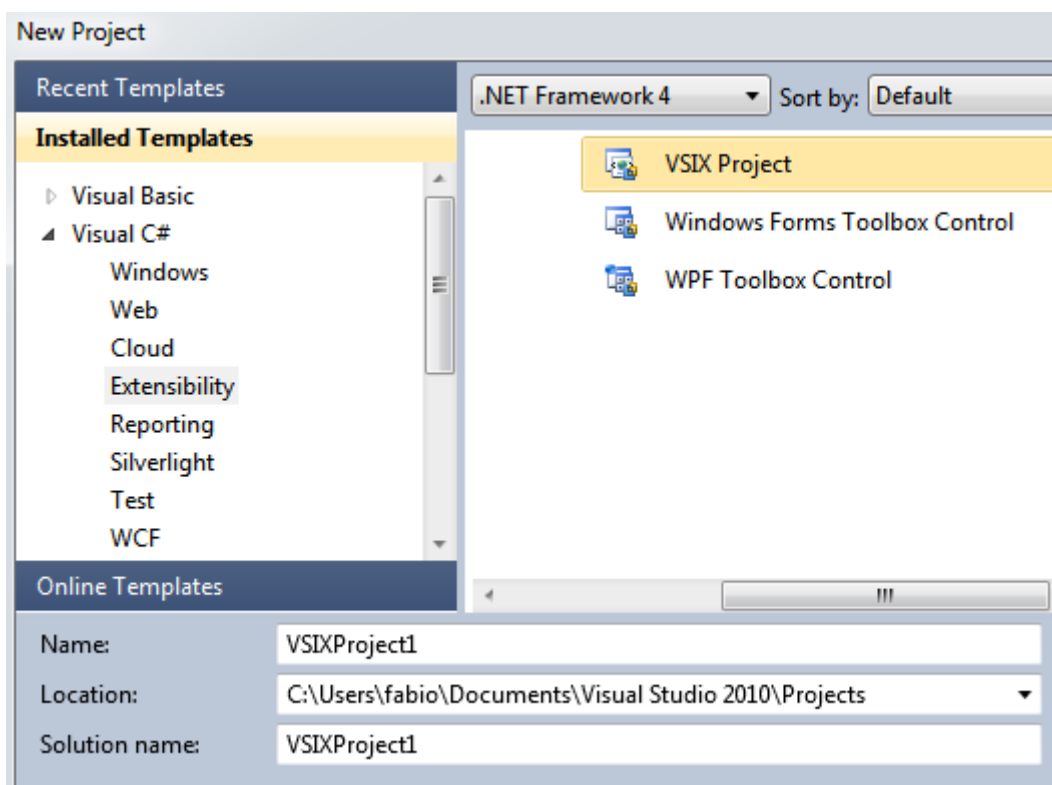


Figura 2.1: creazione di una nuova estensione come VSIX Project

2.3.2 Analisi del Manifest File

Al termine della elaborazione Visual Studio genera un nuovo progetto contenente il solo file “source.extension.vsixmanifest”, chiamato “manifest file” dell'estensione, internamente strutturato come un file XML e contenente le informazioni di seguito riportate, le quali sono facilmente individuabili nella Tabella 2.2:

- nome della estensione;
- stringa identificativa;
- autore;
- versione;
- descrizione;
- versione supportate di Visual Studio;
- versioni supportate del Framework .Net.

Oltre a queste informazioni è presente un prologo con la specifica dello schema xml da utilizzare per interpretare correttamente tutti i parametri.

```
<?xml version="1.0" encoding="utf-8"?>
<Vsix
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  Version="1.0.0"
  xmlns=http://schemas.microsoft.com/developer/vsx-schema/2010
>
  <Identifier Id="VSIXProject1.Microsoft.2e8670d2-a39d-4fc8-968b-
    470872f7a44b">
    <Name>VSIXProject1</Name>
    <Author>Fabio Gravina</Author>
    <Version>1.0</Version>
    <Description xml:space="preserve">Empty VSIX Project.</Description>
    <Locale>1033</Locale>
    <SupportedProducts>
      <VisualStudio Version="10.0">
        <Edition>Ultimate</Edition>
        <Edition>Pro</Edition>
      </VisualStudio>
    </SupportedProducts>
    <SupportedFrameworkRuntimeEdition MinVersion="4.0" MaxVersion="4.0"
    />
  </Identifier>
  <References />
  <Content />
</Vsix>
```

Tabella 2.2: Manifest File di una Estensione

2.4 Template

Il template rappresenta il modello di creazione di un nuovo file in funzione della sua tipologia. All'interno del template trovano posto il contenuto predefinito del file e un insieme di proprietà accessorie che permettono all'ambiente di sviluppo di gestire al meglio i file creati usando il template.

Nel caso specifico del IDE Visual Studio questi modelli di file prendono il nome di “Item Template”.

2.4.1 Creare un nuovo Template Item

Per realizzare un nuovo Template Item è sufficiente creare un nuovo progetto di tipo “Item Template”. Al termine della procedura guidata Visual Studio genera all’interno del progetto tre file:

- modello;
- icona da associare;
- manifest.

All’interno del modello si trova il contenuto predefinito di ogni file creato usando quel particolare template, a titolo di esempio una nuova classe C# viene creata seguendo il seguente modello:

```
using System;
using System.Collections.Generic;
$if$ ($targetframeworkversion$ >= 3.5)
using System.Linq;
$endif
$using System.Text;

namespace $rootnamespace$
{
    class $safeitemrootname$
    {
    }
}
```

Tabella 2.3: Modello di una Classe in linguaggio C#

Il manifest è un file XML che contiene tutte le informazioni necessarie a registrare il nuovo template nella galleria di Visual Studio, cioè nella collezione di tutti i possibili template selezionabili all’atto della creazione di un nuovo Item. Esso è strutturato in due parti: *TemplateData* e *TemplateContent*. Dalla Tabella 2.4 si evince che nella prima sezione vi trovano posto:

- nome, associato al Template Item;

- identificativo alfanumerico univoco, con il quale è possibile identificare il Template Item;
- descrizione, con la quale è possibile aggiungere ulteriori informazioni relative al template;
- icona, associata al Template Item nel menu di Visual Studio;
- nome di Default del nuovo File.

Nella seconda sezione, cioè `TemplateContent`, viene riportata la definizione vera e propria del nuovo file da aggiungere, che contiene il nome del file e il `CustomTool` da associare.

```
<?xml version="1.0" encoding="utf-8"?>
<VSTemplate Version="3.0.0" Type="Item"
  xmlns="http://schemas.microsoft.com/developer/vstemplate/2005">
  <TemplateData>
    <Name>ItemTemplate1</Name>
    <Description>&lt;No description available&gt;</Description>
    <Icon>ItemTemplate1.ico</Icon>
    <TemplateID>c0881b82-4447-4b03-8a68-5c92e505d5a8</TemplateID>
    <ProjectType>CSharp</ProjectType>
    <RequiredFrameworkVersion>2.0</RequiredFrameworkVersion>
    <NumberOfParentCategoriesToRollUp>
      1</NumberOfParentCategoriesToRollUp>
    <DefaultName>Class.cs</DefaultName>
  </TemplateData>
  <TemplateContent>
    <References>
      <Reference>
        <Assembly>System</Assembly>
      </Reference>
    </References>
    <ProjectItem ReplaceParameters="true" CustomTool="" >
      Class.cs</ProjectItem>
    </TemplateContent>
  </VSTemplate>
```

Tabella 2.4: Manifest File di un Template Item

2.5 Generatore di Codice

Un generatore di codice è un componente software realizzato con lo scopo di generare codice sorgente a partire da alcune informazioni chiave fornite dallo sviluppatore.

Esistono diverse tipologie di generatori di codice tra cui:

- generatori di tipo Text Templating, basati cioè su normali file di testo contenenti il modello della classe e il cui funzionamento si basa sulla sostituzione di place holder presenti nel modello coi i valori finali;
- generatori basati sul diagramma delle classe: negli IDE avanzati è possibile trasformare un diagramma UML realizzato tramite tool grafici in una classe;
- generatori basati su xml; in cui la struttura della classe è specificata secondo uno schema xml, che poi viene tradotto nella classe vera e propria.

Il primo meccanismo è language oriented nel senso che è valido solo per un linguaggio in quanto esplicita la struttura della classe seguendo le regole sintattiche del linguaggio scelto. Gli altri due meccanismi tendono invece a specificare solo le proprietà della classe che potrà poi quindi essere implementata in linguaggi diversi.

Visual studio supporta tutte e tre queste tipologie di generatori: in questa tesi viene però usato solo il meccanismo del text templating in quanto è il più indicato per il contesto d'uso.

In Tabella 2.5 è stato riportato un semplice esempio in cui viene illustrato come implementare una delle più comuni funzioni dei più diffusi ambienti di sviluppo: la creazione di una nuova classe con costruttore vuoto.

Tutte le informazioni personalizzabili della classe sono presenti nel template sotto forma di place holder, cioè stringhe segnaposto caratterizzate, in questo caso, dal carattere iniziale %. Dopo che saranno stati inseriti i valori desiderati, ad esempio attraverso una procedura guidata, i place holder saranno sostituiti con il valore corrispondente.

Template	Generated Code
<code>%accessType class %className{ Public %className(){ } }</code>	<code>Public class Person{ Public Person(){ } }</code>

Tabella 2.5: Esempio di generazione di codice. In questo caso in linguaggio C#.

Capitolo 3

Programmazione multi-paradigma in Prolog e .Net

3.1 tuProlog.Net

Lo sviluppo degli strumenti per la programmazione multi-paradigma tra tuProlog e Java è stato facilitato dall'origine stessa del tuProlog, cioè di un interprete Prolog scritto in Java; il che significa che in fondo il tuProlog è codice Java e quindi affinché l'interazione tra i due linguaggi avvenga con successo è necessario affrontare e risolvere i problemi di accesso relativi alle funzionalità e alle risorse, tenendo presente che una volta risolti le risorse saranno direttamente utilizzabili.

Il discorso è molto diverso per ciò che riguarda l'interazione tra tuProlog e i linguaggi del framework .Net. In questo caso anche la più elementare delle strutture dati, o la definizione delle classi avviene secondo approcci diversi e il codice intermedio generato non è compatibile tra i due diversi linguaggi in quanto: nel caso del tuProlog viene generato il bytecode Java, mentre nel caso del .Net viene generato il bytecode CIL.

A seguito di queste profonde diversità sono state portate avanti due differenti versioni del progetto tuProlog: quella di riferimento sviluppata in Java e quella alternativa/secondaria, conosciuta come tuProlog.Net, basata sulla riscrittura del codice Java in C#.

Purtroppo questo approccio implicava di dover lavorare su due fronti rallentando lo sviluppo e creando versioni non allineate dei due rami.

Per ovviare a queste complicazioni si è scelto un approccio differente basato sulla traduzione del codice Java in .Net in modo automatico, appoggiandosi al software IKVM.NET^[16].

3.2 IKVM

IKVM è una implementazione di Java per il framework .Net. Può essere scomposto nelle tre seguenti componenti basi:

- una Java Virtual Machine sviluppata in .Net;
- una libreria Java, derivata dalla riscrittura in .Net della libreria OpenJDK;
- alcuni strumenti aggiuntivi tra cui ikvmc per la traduzione del codice Java in .Net.

IKVM dispone di due modalità di traduzione del codice: statica e dinamica. La modalità statica prevede l'utilizzo del tool ikvmc che genera un assembly incapsulato in file dll o exe; ciò dipende dal fatto che ciò che si sta traducendo sia una libreria o una applicazione. La modalità dinamica implementata, con il tool ikvm, esegue una traduzione a runtime del bytecode Java in .Net che viene immediatamente eseguito.

Utilizzando IKVM, il progetto tuProlog scritto in Java, quando maturo/pronto per un nuovo rilascio, viene semplicemente tradotto in .Net in modo statico, rendendo immediatamente disponibile la nuova versione per entrambe le piattaforme.

3.3 Approcci alla programmazione multi paradigma

L'interoperabilità tra tuProlog e .Net rispecchia i filoni già analizzati nel Capitolo 1.4, infatti si hanno i seguenti possibili scenari:

- utilizzare .Net da tuProlog: OOLibrary
- utilizzare tuProlog da .Net: tuProlog API
- accrescere tuProlog via .Net: nuove librerie
- accrescere .Net via tuProlog: P@J via .NET

3.3.1 Usare .Net da tuProlog: OOLibrary

La nuova tecnica di generazione automatica della versione .Net di tuProlog, ottenuta grazie ad IKVM, fa sì che in questa versione sia anche disponibile la JavaLibrary, cioè la libreria che ci permette di accedere al mondo Java a partire dal codice tuProlog. La JavaLibrary utilizzata da tuProlog.Net ci permette di accedere agli oggetti nativi .Net ma purtroppo solo in modo parziale, perché essendo progettata e realizzata appositamente per Java non riesce a accedere a tutte le peculiarità degli oggetti .Net, come ad esempio le Proprietà di una classe.

Per ovviare a questi problemi, che di fatto limiterebbero le possibilità di utilizzo, è stata sviluppata la libreria OOLibrary che, estendendo la JavaLibrary, è compatibile sia con Java che con .Net.

Infine sono state introdotte le *Conventions*: cioè un meccanismo per gestire le diverse convenzioni sui nomi dei metodi o delle proprietà nei diversi linguaggi. E' naturale aspettarsi che a questo punto sia possibile utilizzare i diversi linguaggi imperativi nel modo più simile/trasparente possibile. Nella Tabella 3.1 vediamo come cambia l'uso della OOLibrary con e senza le convention.

<pre>visualbasicWithoutConvention :- new_object('VBStudent.Student',[123456, john, smith], Obj), Obj <- 'PrintStudent' returns Student, <u>Obj <- get Id returns StudentNumber,</u> class('VBStudent.Student,VBStudent') <- get_StaticProperty returns Value, new_object('VBStudent.Student, VBStudent[]',[10], Array).</pre>
<pre>visualbasicWithConvention :- load_convention('VBConvention.dll','VBConvention.VBDotNet',Conv), new_object(Conv,'VBStudent.Student, VBStudent', [123456, john, smith], Obj), Obj <- printStudent returns Student, <u>Obj.id <- get(StudentNumber),</u> class('VBStudent.Student, VBStudent').staticProperty <- get(Value), new_object(Conv, 'VBStudent.Student, VBStudent()',[10], Array).</pre>
Tabella 3.1: OOLibrary senza e con Convention

3.3.2 Usare tuProlog da .Net: tuProlog API

Come nel caso precedente, anche in questa occasione tuProlog.Net gode della generazione automatica del codice tramite IKVM che, oltre al motore tuProlog e alla JavaLibrary, traduce anche le API che permettono di accedere al tuProlog da Java, e che a loro volta, dopo la traduzione, permettono di accedere da .Net al motore tuProlog. In Tabella 3.2 ne è riportato un esempio applicativo.

```
string th = @"
    any([X|Xs],X,Xs) .
    any([X|Xs],E,[X|Ys]) :- any(Xs,E,Ys) .
    perm([],[]) .
    perm(Xs,[X|Ys]) :- any(Xs,X,Zs), perm(Zs,Ys) .

";
Prolog engine = new Prolog();
Theory t = new Theory(th);
Term goal = new Struct("perm",Term.createTerm(InputTextBox.Text),new
Var("R"));
SolveInfo SI = engine.solve(goal);
string result;
if (!SI.isSuccess())
    result = "no";
while (SI.isSuccess()){
    result = SI.getTerm("R").toString() + "\t";
    if (prolog.hasOpenAlternatives())
        SI = prolog.solveNext();
    else
        break;
}
```

Tabella 3.2: Esempio di utilizzo di tuProlog da .Net

3.3.3 Accrescere tuProlog via .Net: nuove librerie

Lo sviluppo di nuove librerie per tuProlog scritte in .Net ha lo scopo di accrescere le potenzialità del linguaggio tuProlog mettendo a disposizione nuovi predicati utilizzabili nel codice tuProlog, i quali quando vengono richiamati dal motore inferenziale vanno in realtà ad eseguire metodi .Net. Questo approccio permette di usufruire dei vantaggi della programmazione imperativa in ambito tuProlog. Nel

Tabella 3.3 è stata realizzata una libreria .Net che estende Library, con lo scopo di fornire al tuProlog due nuovi predicati: sum e println;

```
using alice.tuprolog;
public class TestLibrary: Library {
    // functor sum(A,B)
    public Term sum_2(Number arg0, Number arg1){
        float n0 = arg0.floatValue();
        float n1 = arg1.floatValue();
        return new Float(n0+n1);
    }
    // predicate println(Message)
    public boolean println_1(Term arg){
        Console.WriteLine(arg);
        return true;
    }
}
```

Tabella 3.3: Accrescere tuProlog via .Net

3.3.4 Accrescere .Net via tuProlog: P@J framework

Per ciò che concerne l'utilizzo del framework P@J dall'ambiente .Net è necessario differenziare due scenari:

- le applicazioni Java che utilizzano P@J funzionano anche dopo esser state tradotte in .Net;
- lo sviluppo di nuove applicazioni .Net, a causa di una eccezione nell'uso del tool Javaassist, non può essere portato a termine correttamente.

Il supporto al framework P@J sarà ulteriormente investigato nel capitolo 4.

Capitolo 4

Approccio basato sull'utilizzo di P@J su .Net

4.1 Stato Attuale

Attualmente, il framework P@J fornisce solo una parte delle sue funzionalità quando utilizzato nell'ambiente .Net. Infatti,

- le applicazioni Java che utilizzano P@J continuano a funzionare anche dopo esser state tradotte in .Net, mentre
- le applicazioni .Net che tentino di utilizzare il framework falliscono. La ragione di fondo per questo comportamento risiede nel diverso modo con cui Java e .NET gestiscono (a compile time, ma soprattutto a run time) i tipi generici, pesantemente utilizzati dal framework P@J per il proprio funzionamento – in particolare riguardo alla type inference. Ciò rende impossibile per la .NET virtual machine di interoperare con le DLL tradotte da IKVM partendo da sorgente Java, in quanto le signature dei metodi non corrispondono a livello di bytecode.

Per questo motivo, in questo capitolo si analizzeranno le differenze di implementazione dei *Generici* tra Java e .Net, per comprendere le cause ultime del problema e valutare le eventuali possibilità di soluzione.

4.2 Tipi Generici

Un tipo generico è un costrutto linguistico che permette di definire altri costrutti in modo indipendente dal tipo specifico su cui saranno chiamati, pur mantenendo un forte controllo di tipo (strong typing) ad operare. In assenza di tali costrutti, la definizione di funzioni e strutture dati generiche passa normalmente per l'uso di un tipo “generale” (come Object nei linguaggi ad oggetti) o finto (come null in C)

aventi l'effetto netto di disabilitare il controllo di tipo, rendendo quella determinata struttura compatibile con qualunque argomento reale effettivamente passato – con ovvie conseguenze negative in termini di affidabilità, possibilità di debugging, chiarezza espressiva, etc, Per questo motivo, i linguaggi a oggetti moderni supportano tutti questa funzionalità, consentendo di definire *metodi generici* e *classi generiche*.

In Tabella 4.1 si riporta un esempio concettuale, di come è possibile creare una classe per la gestione di una lista di elementi in un primo caso utilizzando in tipo generale Object e nel secondo usando i generici.

Nel primo caso si è definita una classe contenente una lista di Object, questo ci obbliga a fare il cast dal tipo di interesse (int) verso Object prima di poter aggiungere l'elemento alla lista; allo stesso modo dovremo fare un cast da Object a int dopo aver estratto l'elemento dalla lista. Nel secondo caso la classe, e la signature del metodo, sono rese parametriche rispetto al tipo, T, che potrà essere istanziato a runtime, senza la necessità di nessun cast.

<pre>public class ListOfObject{ private int list[]; private int top; public void Add(Object a){ list[top] = a; top++; } }</pre>	<pre>public class ListOf<T>{ private T list[]; private int top; public void Add(T a){ list[top] = a; top++; } }</pre>
<pre>ListOfObject i_list = new ListOfObject(); i_list.Add((Object)3); ListOfObject d_list = new ListOfObject(); d_list.Add((Object)3.0);</pre>	<pre>ListOf<int> i_list = new ListOf<int>(); i_list.Add(3); ListOf<double> d_list = new ListOf<double>(); d_list.Add(3.0);</pre>
Tabella 4.1: Generalizzazione attraverso Object e generici	

4.3 Tipi Generici in Java e .Net

Entrambe le piattaforme Java e .Net supportano i tipi generici, con le stesse finalità d'uso e con una sintassi molto simile, ma con notevoli differenze implementative poiché nel caso di Java sussisteva il vincolo di non modificare la macchina virtuale, definita in origine senza tenere conto di tale concetto.

4.3.1 Implementazione dei Tipi Generici in Java

In ambito Java si è scelto un approccio basato sul meccanismo della *Type Erasure*, letteralmente cancellazione del tipo, perché questo meccanismo permette di non modificare la macchina virtuale Java. La *type erasure* si applica a compile time e ciò significa che quando il codice di classi o metodi generici viene compilato i tipi generici vengono rimossi e sostituiti con il tipo generale `Object`. Considerando che in Java ogni oggetto deriva da `Object` sarà possibile passare come parametro alla funzione `print` un oggetto di qualsiasi tipo. E' anche possibile specificare dei vincoli (*Constraint*) per un tipo generico, in questo modo si fa sì che quando si crea la classe generica tutti i tipi passati per parametro debbano rispettare tali vincoli.

Si veda nell'esempio in Tabella il tipo generico `U` nel quale si impone che il tipo passato per parametro estenda la classe `Number`. In tal caso nel bytecode generato `U` non sarà sostituito con `Object` ma bensì con `Number` così da rispettare il vincolo anche a runtime. Questo approccio implementativo dei generici non consente di effettuare alcune operazioni, tra le quali quelle evidenziate nel metodo `demo`. Le operazioni non consentite sono in particolare quelle in cui è necessario instanziare i generici.

<pre> class GenericPrint <T, U extends Number>{ private T x; private U u; public void print(T y){ System.out.print(y); } Public void demo(){ T k = new T (); // error T[] ar = new T [0]; // error Class c = T.class; // error } } </pre>	<pre> class GenericPrint{ private Object x; private Number u; public print(Object y){ System.out.println(y); } } </pre>
Tabella 4.2: Implementazione Java dei generici	

4.3.2 Implementazione dei Tipi Generici in .Net

L'implementazione dei tipi generici in .Net è passata per un approccio diverso; in questa piattaforma i generici sono stati implementati a livello di intermediate language. Ciò significa che l'intermediate language è stato modificato per supportarli nativamente e di conseguenza è stata modificata anche la macchina virtuale .Net.

Questo approccio, che quindi fornisce un supporto nativo ai generici, fa sì che questi restino reificati a runtime; come è possibile osservare in Tabella 4.3 il codice compilato è rimasto generico rispetto al tipo T. Il metodo `demo` ha subito alcune variazioni in fase di compilazione per consentire di portare a buon fine le operazioni di istanziazione dei generici.

<pre> class GenericClass<T> where T : new(){ private T x; public void print(T y){ Console.Write(y); } Public void demo(){ T k = new T(); // OK with new() constraint. T[] ar = new T [0]; // OK Type type = typeof(T); // OK } } </pre>
<pre> internal class GenericClass<T> where T : new(){ private T x; public GenericClass(){} public void Print(T y){ Console.Write(y); } public void demo(){ T t; T t1 = default(T); if (t1 == null){ t = Activator.CreateInstance<T>(); } else{ t1 = default(T); t = t1; } T k= t; T[] ar = new T[0]; Type type = typeof(T); } } </pre>
Tabella 4.3: Generici in .Net, esempio riportato in linguaggio C#

4.4 Considerazioni su P@J in .Net

4.4.1 P@J e i generici

Dopo aver analizzato le differenze implementative dei generici tra .Net e Java si può passare all’analisi dei due casi specifici inerenti P@J.

Le applicazioni scritte in Java che fanno uso di P@J continuano a funzionare anche dopo la traduzione del codice, perché in tal caso il meccanismo dei generici è utilizzato da due entità Java e quindi i generici sono gestiti allo stesso modo. Quando il bytecode java viene tradotto in .Net da IKVM i generici sono stati già correttamente gestiti e quindi non si ha nessun errore.

Differente è la situazione per le applicazioni scritte in .Net nelle quali il meccanismo dei generici viene usato cross-language; ovvero vengono richiamati in .Net e risolti in Java. In Tabella 4.4 si può osservare come, se funzionasse, dovrebbe essere definito un PrologMethod P@J lato .Net e come lo si dovrebbe poter utilizzare.

```
using List = alice.tuprologx.pj.model.List;
public abstract class Perm{
    [PrologMethodAttribute(clauses = new string[]{
        "permutation([],[]).",
        "permutation(U, [X|V]) :- remove(U,X,Z), permutation(Z,V).",
        "remove([X|T], X, T).",
        "remove([X|U], E, [X|V]) :- remove(U,E,V). "
    })]
    public abstract Y permutation<X,Y>(<X> l)
        where X : List
        where Y : List;
}

public class Program{
    static void Main(string[] args){
        ...
        List list = new List(1);
        java.lang.Class c =
            java.lang.Class.forName("javaprolog.Perm, javaprolog");
        Perm perm = (Perm) PJ.newInstance(c);
        List retList = perm.permutation<List,List>(list);

        Console.WriteLine(retList.toJava());
    }
}
```

Tabella 4.4: Esempio di codice .Net che usa P@J

Tenendo in considerazione le differenze implementative dei generici in Java e .Net, ci si aspetta che il codice appena illustrato venga compilato correttamente ma che riporti degli errori a runtime, in quanto il meccanismo di Type Erasure fa sì che a runtime i tipi di ritorno non coincidano con quanto atteso da .NET.

In effetti, l'esecuzione del codice di esempio restituisce la seguente eccezione:

```
AbstractMethodError was unhandled
```

```
Method javaprolog.Perm.permutation is unsupported by IKVM
```

L'eccezione, stata generata dal tool IKVM, appare confermare l'ipotesi precedente. Come ulteriore verifica è stato contattato il team di sviluppo di IKVM, che ha confermato che l'ipotesi sopra descritta è effettivamente corretta e che l'impossibilità di recuperare il metodo "annotato da Java" è dovuta non tanto a una limitazione di IKVM in sé, ma, appunto, alla diversa gestione dei tipi generici lato Java, che cancella a livello di bytecode informazioni cruciali per il successivo lookup del metodo in .NET.

Scemano così le possibilità di far funzionare il framework P@J rispettando uno degli obbiettivi prefissati, cioè non modificare l'attuale implementazione java del framework: la sola possibilità teorica di proseguire per questa strada, infatti, implicherebbe di rivedere profondamente il funzionamento del framework P@J, riprogettando le parti relative alla gestione dei tipi. Considerato però la lunga attività di ricerca necessaria in passato per giungere all'attuale impostazione, tale via non appare percorribile.

Pertanto, nel capitolo successivo verrà studiato un diverso approccio, che consenta di raggiungere ugualmente il risultato per una strada diversa, nel rispetto del vincolo di non modificare l'attuale progetto (e codice) di P@J.

Capitolo 5

Approccio basato sui generatori di codice per .Net

In questo capitolo verrà descritta una diversa metodologia di integrazione tra tuProlog e .Net, basata sulla generazione di codice sorgente .Net a partire da una teoria tuProlog. Tale generazione del codice .Net sarà implementata utilizzando un generatore di codice. Sarà così possibile sfruttare le potenzialità di tuProlog utilizzando la classe .Net generata.

5.1 Requisiti

1. il generatore di codice deve poter funzionare a partire da sorgenti tuProlog standard, cioè senza che sia necessario modificarli per renderli compatibile con il generatore stesso;
2. il generatore di codice deve consentire la personalizzazione del nome della classe generata e del namespace;
3. il generatore di codice deve offrire il supporto al linking statico e dinamico del codice tuProlog. Deve quindi offrire le due seguenti modalità di funzionamento:
 - 3.1. statica: il codice tuProlog è incapsulato all'interno del codice .Net all'atto della generazione della classe;
 - 3.2. dinamica: il codice tuProlog viene caricato nella classe .Net a runtime consentendo successiva la modifica del codice tuProlog senza la necessità di rigenerare la classe.

5.2 Analisi dei requisiti

Per soddisfare i requisiti appena descritti è necessario effettuare alcune scelte di progetto che saranno di seguito illustrate.

I linguaggi .Net ai quale si vuole fornire supporto implementativo sono il C# e VisualBasic, in quanto i più diffusi di questo framework. Dato che i due linguaggi sono entrambi object oriented e differiscono solo per la notazione sintattica, nel resto del capitolo vengono ci si riferirà solo al C#; fermo restando che fatte le dovute analogie sintattiche tutto ciò che sarà discusso resta valido anche per il VisualBasic.

5.2.1 Modello della classe generata

Affinché sia possibile generare una classe C# fully qualified è necessario specificare nel file generato il namespace, cioè lo spazio dei nomi a cui afferirà la classe, e il nome della classe stessa. Per soddisfare questa necessità e al contempo il requisito 1 si è scelto di:

- assegnare al namespace lo stesso nome del progetto .Net all'interno del quale si trova il file tuProlog in cui si troverà la classe generata;
- assegnare alla classe lo stesso nome del file tuProlog, ovviamente sprovvisto di estensione.

Nella Tabella 5.1 viene illustrato tramite un esempio quello che sarà il codice generato come comportamento di default del sistema.

Perm.pl	Perm.cs
<pre> any([X Xs],X,Xs). any([X Xs],E,[X Ys]) :- any(Xs,E,Ys). permutation([],[]). permutation(Xs,[X Ys]) :- any(Xs,X,Zs), permutation(Zs,Ys). </pre>	<pre> using System; using System.Collections.Generic; using System.Linq; using System.Text; using alice.tuprolog; namespace PrologTestSolution { public class Perm : Prolog { private string th = @" any([X Xs],X,Xs). any([X Xs],E,[X Ys]) :- any(Xs,E,Ys). perm([],[]). perm(Xs,[X Ys]) :- any(Xs,X,Zs), perm(Zs,Ys). "; public Perm(){ setTheory(new Theory(th)); } } } </pre>
Tabella 5.1: Generazione di una classe C# non personalizzata	

5.2.2 Personalizzazione della classe generata

Il requisito 2 richiede di fornire allo sviluppatore la possibilità di personalizzare i due parametri precedenti, si rende necessario annotare il codice tuProlog con meta-informazioni, che saranno usate dal generatore di codice:

```
%NameSpace nomeDelNameSpace
```

```
%ClassName nomeDellaClasse
```

le meta-info per il generatore devono ovviamente essere viste come commenti dal motore prolog per non interferire con esso, quindi l'aggiunta di queste keyword non modifica in nessun modo la logica del codice tuProlog.

5.2.3 Modalità di generazione

Per dare una maggiore flessibilità al programmatore il requisito 2 richiede due modalità di generazione del codice attraverso le quali è possibile scegliere se incorporare la teoria all'interno della classe, oppure se mantenere la teoria in un file esterno accessibile al software. Le due modalità prendono rispettivamente il nome di modalità Statica e Dinamica.

Il comportamento di default del generatore segue le regole della modalità statica, per attivare la modalità dinamica è necessario annotare il sorgente tuProlog con la seguente meta-informazione:

```
%ExtFile nomeFileProlog.pl
```

5.2.3.1 Modalità statica

Questa modalità si basa sull'idea di portare dentro il codice .Net la teoria tuProlog in quanto questa è semplicemente un stringa e quindi facile da gestire. Il risultato di questo incapsulamento è un codice chiuso, cioè dà meno possibilità allo sviluppatore o utilizzatore di andare ad agire sulla teoria dopo il processo di generazione. Questa rigidità porta dei vantaggi in termini di sicurezza e robustezza del software in quanto siamo sicuri che il comportamento della classe generata sarà sempre lo stesso. Di contro vengono limitate: le personalizzazioni post-generazione che dovranno passare attraverso un processo di estensione della classe generata; e le correzioni della teoria tuProlog che richiederanno la ricompilazione dell'intero progetto. Il codice generato utilizzando questa modalità è lo stesso di quello visto in Tabella 5.1.

5.2.3.1 Modalità dinamica

Per ovviare ai limiti della modalità statica, accettando al contempo un compromesso sulla intrinseca sicurezza derivante da un codice chiuso e sulle prestazioni, è stata introdotta la modalità dinamica. Utilizzando questa modalità nel file tuProlog non specificheremo più il sorgente tuProlog ma bensì il file in cui andare a recuperarlo e il cui contenuto sarà caricato a runtime dal software. In

questo modo sarà possibile modificare la teoria senza ricompilare il software ma solo aggiornando il file della teoria. Gli svantaggi di questa tecnica sono principalmente legati a due possibili situazioni: la cancellazione accidentale o volontaria del file tuProlog, che rende il software inutilizzabile; e la volontaria manomissione dello stesso da parte di male intenzionati. Si veda a corredo l'esempio in Tabella 5.2 inerente la risoluzione simbolica delle derivate, il cui codice tuProlog si suppone sia contenuto nel file Der.pl. Per il contenuto di questo file si veda l'esempio completo nel Capitolo 6.

<pre>%ClassName PrologDerivate %ExtFile Der.pl</pre>	<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; using alice.tuprolog; using java.io; namespace PrologTestSolution { public class PrologDerivate : Prolog { public PrologDerivate(){ setTheory(new Theory(new FileInputStream("Der.pl"))); } } }</pre>
Tabella 5.2: Generazione di una classe C# in modalità dinamica	

5.2.3.3 Confronto Prestazionale

Dal punto di vista prestazionale le differenze sono minime in quanto la differente modalità di caricamento della teoria influenza soltanto la creazione dell'oggetto. Una volta portato a termine il costruttore i due oggetti saranno equivalenti e quindi anche le prestazioni si equivarranno. Questa affermazione è vera a patto che non ci sia la necessità di instanziare molti oggetti della stessa classe, caso poco comune dato che tutti gli oggetti fornirebbero la medesima funzionalità.

5.3 Progetto

Per realizzare il generatore di codice descritto nei requisiti si è scelto di realizzare il progetto sotto forma di estensione per Visual Studio piuttosto che come tool esterno.

Questa scelta è giustificata dal fatto che l'integrazione con Visual Studio rende tutto il processo di sviluppo dell'applicazione più lineare e nasconde all'utilizzatore la presenza del generatore di codice. Un ulteriore vantaggio di questa metodologia risiede nella possibilità di utilizzare un canale di comunicazione, messo a disposizione dal meccanismo delle estensioni, attraverso il quale il generatore di codice può inviare messaggi inerenti l'esito della generazione del codice a Visual Studio che si occuperà di visualizzarle opportunamente.

5.3.1 Vincoli progettuali

In riferimento a quelle che saranno le scelte progettuali e implementative è doveroso specificare che alcune di queste derivano direttamente da vincoli imposti dal meccanismo di gestione delle estensioni, dalla interfaccia `IVsSingleFileGenerator` e dalle classi astratte `BaseCodeGenerator` e `BaseCodeGeneratorWithSite`.

5.3.2 Diagramma delle Classi

Nel diagramma delle classi sottostante (Figura 5.3) è possibile vedere come il `PrologGenerator` sia correlato al resto del sistema.

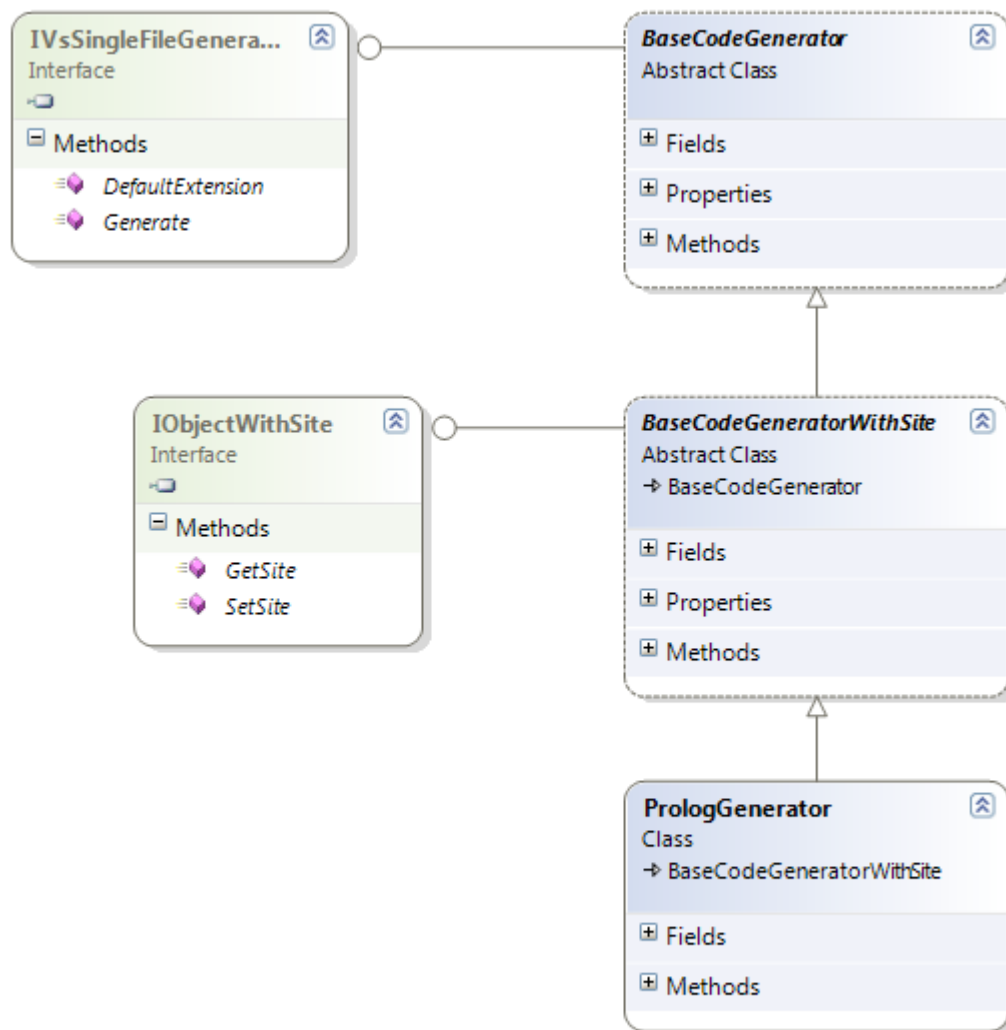


Figura 5.3: Diagramma delle Classi

5.3.2.1 IVsSingleFileGenerator

Questa interfaccia appartenente al Visual Studio Development Kit definisce i metodi che un generatore di file deve implementare.

Il metodo *DefaultExtension* restituisce, attraverso il parametro di out, l'estensione che dovrà essere assegnata al file generato (il nome del file sarà lo stesso di quello di partenza); il metodo stesso restituirà come valore di ritorno: `VSConstant.S_OK` in caso di successo oppure `VSConstant.E_FAIL` in caso di errore.

```
int IVsSingleFileGenerator.DefaultExtension(out string pBstrDefaultExtension)
```

Il metodo *Generate* accetta come parametri di input:

- `wszInputFilePath`: il percorso del file originale;
- `bstrInputFileContents`: il contenuto del file originale;
- `wszDefaultNameSpace`: il NameSpace in cui sarà salvato il file;
- `pGenerateProgress`: un riferimento alla interfaccia `IVsGeneratorProgress` per gestire l'avanzamento della generazione;

e restituisce:

- `rgbOutputFileContents`: il contenuto del file generato;
- `pcbOutput`: il numero di byte contenuti in `rgbOutputFileContents`;
- come valore di ritorno: `VSCONSTANT.S_OK` in caso di successo oppure `VSCONSTANT.E_FAIL` in caso di errore.

```
int IVsSingleFileGenerator.Generate(
    string wszInputFilePath, string bstrInputFileContents,
    string wszDefaultNamespace, IntPtr[] rgbOutputFileContents,
    out uint pcbOutput, IVsGeneratorProgress pGenerateProgress )
```

5.3.2.2 BaseCodeGenerator

Questa classe astratta implementa l'interfaccia `IVsSingleFileGenerator` e quindi i due metodi descritti nel paragrafo precedente, aggiunge inoltre i seguenti:

- metodi astratti:
 - `byte[] GenerateCode(string inputFileContent)`: metodo che quando implementato provvederà a generare il contenuto del file di destinazione;
 - `string GetDefaultExtension()`: metodo che quando implementato restituirà l'estensione del file di destinazione;

- metodi implementati:
 - `GenerateError(uint level, string message, uint line, uint column)`: questo metodo permette di segnalare a VisualStudio che si è verificato un errore durante l'esecuzione, inviando: il livello di errore, il messaggio, la linea e la colonna in cui l'errore si è verificato;
 - `GenerateWarning(uint level, string message, uint line, uint column)`: questo metodo permette di segnalare a VisualStudio che la presenza di un warning generato durante l'esecuzione, inviando: il livello di warning, il messaggio, la linea e la colonna relativi al warning stesso;
- proprietà:
 - `FileNameSpace, InputFilePath, HasError`.

5.3.2.3 BaseCodeGeneratorWithSite

La classe astratta `BaseCodeGeneratorWithSite` estende la classe `BaseCodeGenerator` e ne implementa solo il metodo `GetDefaultExtension`. Implementa anche l'interfaccia `IObjectWithSite`, grazie alla quale sarà possibile accedere a molte informazioni inerenti il progetto VisualStudio, all'interno del quale sta eseguendo il generatore. In particolare con il metodo `GetProject()` viene restituito un riferimento al progetto dal quale è possibile estrarre informazioni tra cui: il nome, il percorso e il tipo (VB o C#).

5.4 Implementazione di PrologVSExtension:

L'estensione realizzata è stata suddivisa in due parti: il `PrologGenerator` e il `PrologCSharpItemTemplate`. Entrambe queste porzioni di codice sono incapsulate secondo le metodologie appena viste in due progetti, i quali sono a loro volta incapsulati in una Visual Studio Solution.

- PrologVSExtension (Solution)
 - PrologItemTemplate (ItemTemplate Project)
 - PrologItemTemplate.ico
 - PrologItemTemplate.vstemplate
 - PrologSource.pl
 - PrologGenerator (VSIX Project)
 - PrologGenerator C# Files
 - Source.extension.vsixmanifest
 - Resources Files

5.4.1 PrologItemTemplate

Come già accennato nel paragrafo 2.4 inerente i Template Item, il PrologTemplateItem implementato mediante un ItemTemplate Project, contiene i seguenti file:

- PrologItemTemplate.ico : contenente l'icona che verrà associata ai file prolog;
- PrologItemTemplate.vstemplate: come già visto nel paragrafo 2.4.1, questo file contiene tutti i parametri dell'ItemTemplate tra cui: nome, descrizione, tipo del progetto compatibile;

<pre> <?xml version="1.0" encoding="utf-8"?> <VSTemplate Version="3.0.0" Type="Item" xmlns="http://schemas.microsoft.com/developer/vstemplate/2005"> <TemplateData> <Name>Prolog Item Template</Name> <Description>Prolog File</Description> <Icon>PrologTemplateIcon.ico</Icon> <TemplateID>4c926538-a48b-40c0-9e7f-e3c96f1ff9c5</TemplateID> <ProjectType>CSharp</ProjectType> <RequiredFrameworkVersion>2.0</RequiredFrameworkVersion> <NumberOfParentCategoriesToRollUp>1 </NumberOfParentCategoriesToRollUp> <DefaultName>PrologSource.pl</DefaultName> </TemplateData> <TemplateContent> <ProjectItem ReplaceParameters="true" CustomTool="PrologGenerator"> PrologSource.pl </ProjectItem> </TemplateContent> </VSTemplate> </pre>
Tabella 5.4: Manifest File del PrologItemTemplate

- PrologSource.pl: il contenuto di default del sorgente tuProlog contiene solo la definizione dei due parametri opzionali NameSpace e ClassName già dettagliati nel paragrafo 5.2.1, si noti l’assenza della keyword %ExtFile che indirizza l’utente verso un utilizzo statico della estensione.

<pre> %NameSpace MyNameSpace %ClassName MyClassName </pre>
Tabella 5.5: Contenuto del file PrologSource.pl

5.4.2 PrologGenerator

Questo progetto contiene i file che realizzano il generatore di codice. Il PrologGenerator una volta attivato da Visual Studio procederà alla generazione del codice seguendo tre passi di elaborazione:

- individuazione del tipo di progetto (C# o VB);
- analisi del sorgente tuProlog al fine di individuare eventuali keyword, con particolare attenzione alla keyword %ExtFile che stabilirà quale modello di classe sarà utilizzato;
- generazione della classe personalizzata in funzione delle keyword.

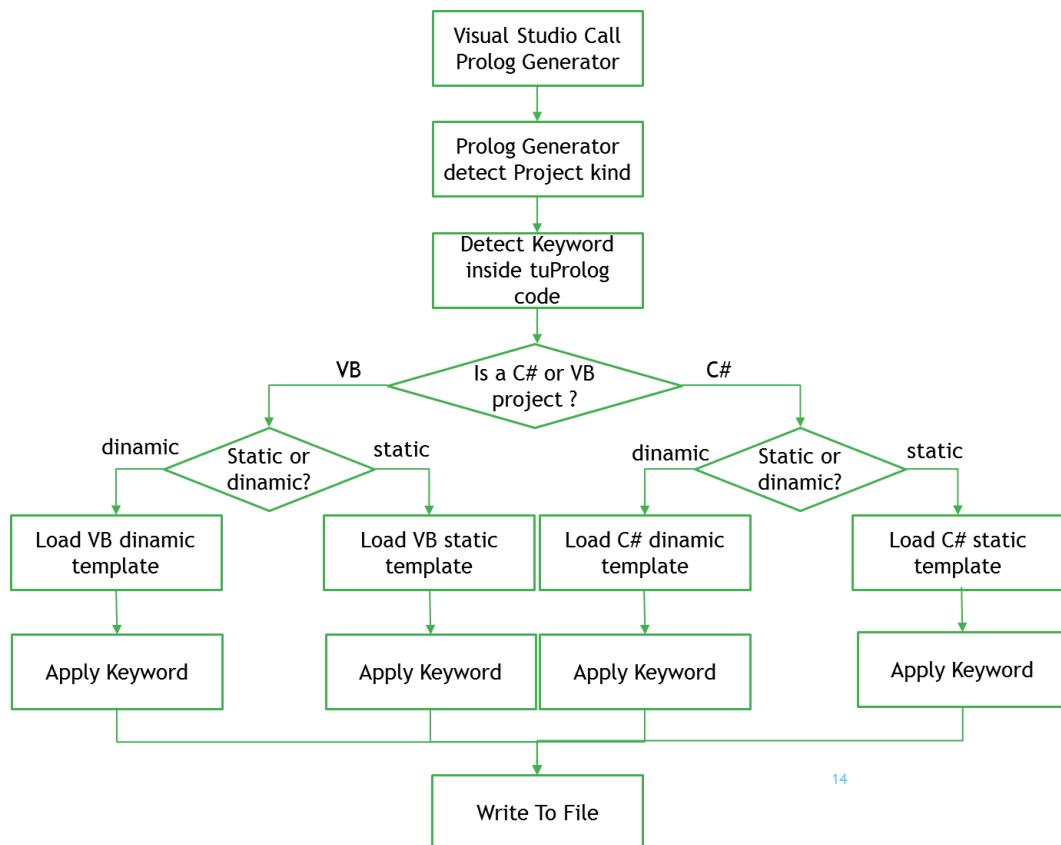


Figura 5.6: Diagramma di Flusso del PrologGenerator

5.4.2.1 Modelli di classe statico e dinamico

In funzione della modalità di generazione della classe che sarà attivata il generatore utilizzerà un diverso modello di classe, questi modelli sono specificati nei file:

CSharpTemplate.txt (Tabella 5.7), per il linking statico; e CSharpExternalTemplate.txt (Tabella 5.8), per il link dinamico.

Gli elementi variabili della classe, cioè: il NameSpace, il nome della classe, la teoria e il nome del file esterno, sono presenti in questi file come place holder, affinché sia possibile sostituirle con i valori corretti derivanti dalla analisi del sorgente tuProlog.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using alice.tuprolog;

namespace #NameSpace {
    public class #ClassName : Prolog {
        private string th = @"
            #Theory";
        public #ClassName() {
            setTheory(new Theory(th));
        }
    }
}
```

Tabella 5.7: Template per generare una classe C# in modalità statica

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using alice.tuprolog;
using java.io;

namespace #NameSpace {
    public class #ClassName : Prolog {
        public #ClassName() {
            setTheory(new Theory(new FileInputStream("#ExtFile")));
        }
    }
}
```

Tabella 5.8: Template per generare una classe C# in modalità dinamica

Verrà adesso analizzato il file principale del progetto, PrologGenerator.cs, suddiviso per sezioni funzionali.

5.4.2.2 Registrazione del COM

La registrazione dell'oggetto COM nel registro di sistema viene realizzata attraverso la specifica dei seguenti attributi della classe:

- Visibilità del COM;
- Guid;
- Registrazione del COM per i progetti C#;
- Registrazione del COM per i progetti VB;
- Indicazione del tipo di oggetto che fornisce le funzionalità, nel nostro caso PrologGenerator stesso.

```
[ComVisible(true)]  
[Guid("52B316AA-1997-4c81-9969-83604C09EEB4")]  
[CodeGeneratorRegistration(typeof(PrologGenerator),  
"C# Prolog Class Generator",  
vsContextGuids.vsContextGuidVCSPProject,  
GeneratesDesignTimeSource=true)]  
  
[CodeGeneratorRegistration(typeof(PrologGenerator),  
"VB Prolog Class Generator",  
vsContextGuids.vsContextGuidVBProject,  
GeneratesDesignTimeSource = true)]  
[ProvideObject(typeof(PrologGenerator))]
```

Tabella 5.9: Registrazione del COM nel registro di sistema

5.4.2.3 Generazione del codice differenziata per tipo di progetto

Il metodo GenerateCode richiamato da Visual Studio, quando quest'ultimo attiva il CustomTool, analizza il tipo di progetto e di conseguenza diversifica la generazione di codice C# da codice VB. Si veda la Tabella 5.10 per la sua implementazione.

```
protected override byte[] GenerateCode(string inputFileContent)
{
    Project project = this.GetProject();
    if (project.FullName.EndsWith(".csproj"))
        return GenerateCSharpCode(inputFileContent);
    else
        return GenerateVBCode(inputFileContent);
}
```

Tabella 5.10: Identificazione del tipo di progetto

5.4.2.4 Generazione di Codice Specifico

La generazione del codice specifico per un determinato linguaggio è eseguita rispettivamente dai metodi: `GenerateCSharpCode` e `GenerateVBCode`. Di seguito, in Tabella 5.11, è riportato il codice solo del metodo `GenerateCSharpCode` in quanto il codice del metodo `GenerateVBCode` risulta speculare.

```

private byte[] GenerateCSharpCode(string inputFileContent){
    StringReader stringReader = new StringReader(inputFileContent);
    List<string> rows = new List<string>();
    List<string> potentialParam = new List<string>();
    string tmp;
    while ((tmp = stringReader.ReadLine()) != null){
        if(tmp.StartsWith("%"))
            potentialParam.Add(tmp.Trim());
        rows.Add(tmp);
    }

    try{

        this.HasError = false;

        // Set optional params if not found set default Value
        string nameSpace, className, extFile;

        extFile = GetValue("%extfile", potentialParam, "");

        nameSpace = GetValue("%namespace", potentialParam,
base.GetProject().Name);

        className = GetValue("%classname", potentialParam,
Path.GetFileNameWithoutExtension(base.InputFilePath));

        string template;
        // if no extfile defined
        if(extFile == ""){
            template = FileResource.CSharpTemplate;
            template = template.Replace("#NameSpace", nameSpace.Trim());
            template = template.Replace("#ClassName", className.Trim());

            // copy and set theory inside template row by row
            string theory = rows.Aggregate("", (current, row) =>
current + (row + "\r\n"));

            template = template.Replace("#Theory", theory);
        }
        else // extFile defined
        {
            template = FileResource.CSharpExternalTemplate;
            template = template.Replace("#NameSpace", nameSpace.Trim());
            template = template.Replace("#ClassName", className.Trim());
            template = template.Replace("#ExtFile", extFile.Trim());
        }

        MemoryStream ms = new MemoryStream();
        StreamWriter writer = new StreamWriter(ms, new UTF8Encoding());
        writer.Write(template);
        writer.Flush();
        ms.Position = 0;

        return this.HasError ? null : ms.ToArray();
    }
    catch (Exception e){
        this.GeneratorError(4, e.ToString(), 1, 1);
        //Returning null signifies that generation has failed);
        return null;
    }
}

```

Tabella 5.11: Riconoscimento e applicazione delle keyword

Capitolo 6

Collaudo della tecnologia

In questo capitolo sarà analizzata una applicazione di test utilizzata per verificare il funzionamento del generatore di codice realizzato e per confrontare l'utilizzo delle due diverse modalità di linking (statica e dinamica) tra loro.

Per realizzare l'applicazione si è scelto di il linguaggio C# anche se è possibile ripetere il procedimento in maniera analoga in VB.

L'applicazione risulterà perciò strutturata in due parti: una logica che si occupa della interazione con il motore tuProlog; e una imperativa che invece si occupa della gestione della interfaccia grafica.

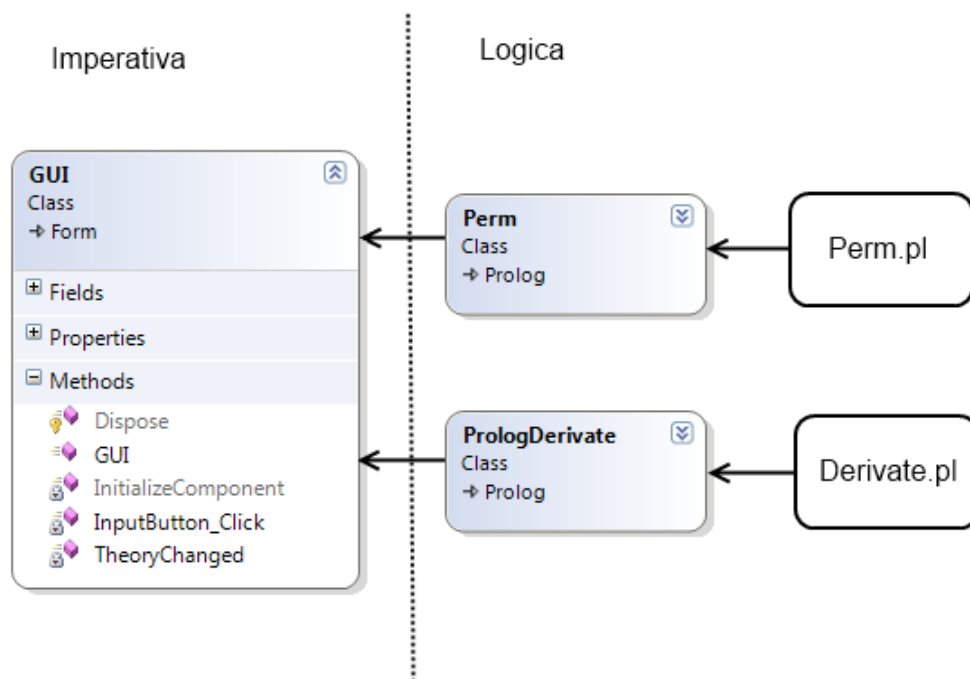


Figura 6.1: Struttura della applicazione

6.1 Sezione Logica

Questa sezione sarà analizzata in due sotto parti afferenti ai due diversi esempi di generazione del codice. Al fine di mostrare entrambe le modalità, statica e dinamica, in questo esempio scegliamo risolvere il problema del calcolo delle permutazioni con la modalità statica e il problema della risoluzione simbolica delle derivate in modalità dinamica, questa è solo una possibilità perché entrambi i problemi sono risolvibili con entrambe le modalità.

6.1.1 Permutazioni

In questo caso l'obiettivo è al risoluzione del calcolo delle permutazioni utilizzando l'algoritmo tuProlog contenuto nel file Perm.pl; come detto sopra si è scelta per questo esempio la modalità statica, quindi si può osservare nel sorgente tuProlog l'assenza della %ExtFile.

Perm.pl
<pre>any([X Xs],X,Xs) . any([X Xs],E,[X Ys]) :- any(Xs,E,Ys) . perm([],[]) . perm(Xs,[X Ys]) :- any(Xs,X,Zs), perm(Zs,Ys) .</pre>
Tabella 6.2: Sorgente tuProlog per la risoluzione delle permutazioni

Inoltre non state applicate neanche le altre keyword progettate nel sistema, a seguito di ciò, l'esecuzione del generatore di codice produce un file C# in cui vengono assegnati valori di default per il Namespace e il ClassName. Si può infatti notare:

- il nome della classe corrispondente al nome del file;
- il namespace uguale al nome del progetto;
- l'utilizzo della modalità statica che ha incapsulato la teoria dentro il file C#.

Perm.cs
<pre> using System; using System.Collections.Generic; using System.Linq; using System.Text; using alice.tuprolog; namespace PrologTestSolution { public class Perm : Prolog { private string th = @" any([X Xs],X,Xs). any([X Xs],E,[X Ys]) :- any(Xs,E,Ys). perm([],[]). perm(Xs,[X Ys]) :- any(Xs,X,Zs), perm(Zs,Ys). "; public Perm(){ setTheory(new Theory(th)); } } } </pre>
Tabella 6.3: Codice C# generato in modalita statica

6.1.2 Derivate

Il questo secondo esempio lo scopo è la risoluzione simbolica delle derivate, utilizzando l’algoritmo tuProlog contenuto nel file Der.pl, volendo testare il funzionamento della modalità dinamica è stato creato il file di “linking” Derivate.pl.

Derivate.pl
<pre> %NameSpace DerivateNamespace %ClassName PrologDerivate %ExtFile Der.pl </pre>
Tabella 6.4: File di collegamento a Der.pl per il linking dinamico

In questo caso sono state utilizzate tutte e tre le keyword progettate e di conseguenza è possibile vedere i loro effetti nel file generato Derivate.cs:

- Namespace e il nome della classe sono quelli specificati nel file tuProlog e non i valori di default;
- la keyword ExtFile ha attivato la modalità dinamica portando all'implementazione di un costruttore che non include la teoria ma che la andrà a leggere a runtime.

Derivate.cs
<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; using alice.tuprolog; using java.io; namespace DerivateNamespace { public class PrologDerivate : Prolog { public PrologDerivate(){ setTheory(new Theory(new FileInputStream("Der.pl"))); } } }</pre>
Tabella 6.5: Classe C# generata in modalità dinamica

La teoria tuprolog sarà contenuta nel file Der.pl che si riporta di seguito per completezza.

```

dExpr(T,DT) :- dTerm(T,DT).
dExpr(E+T,[DE+DT]) :- dExpr(E,DE), dTerm(T,DT).
dExpr(E-T,[DE-DT]) :- dExpr(E,DE), dTerm(T,DT).
dTerm(F,DF) :- dFactor(F,DF).
dTerm(T*F,[DT*F]+[T*DF]) :- dTerm(T,DT), dFactor(F,DF).
dTerm(T/F,[F*DT]-[T*DF]/[F*F]) :- dTerm(T,DT), dFactor(F,DF).
dFactor(x,1).
dFactor(N,0) :- number(N).
dFactor([E],DE) :- dExpr(E,DE).
dFactor(-E,-DE) :- dExpr(E,DE).
dFactor(sin(E), [cos(E)*DE]) :- dExpr(E,DE).
dFactor(cos(E), [-sin(E)*DE]) :- dExpr(E,DE).
dFactor(E^N, [N*E^(Nm1)*(DE)]) :- dExpr(E,DE), Nm1 is N-1.
dExpr(T,DT) :- dTerm(T,DT).
dExpr(E+T,[DE+DT]) :- dExpr(E,DE), dTerm(T,DT).
dExpr(E-T,[DE-DT]) :- dExpr(E,DE), dTerm(T,DT).
dTerm(F,DF) :- dFactor(F,DF).
dTerm(T*F,[DT*F]+[T*DF]) :- dTerm(T,DT), dFactor(F,DF).
dTerm(T/F,[F*DT]-[T*DF]/[F*F]) :- dTerm(T,DT), dFactor(F,DF).
dFactor(x,1).
dFactor(N,0) :- number(N).
dFactor([E],DE) :- dExpr(E,DE).
dFactor(-E,-DE) :- dExpr(E,DE).
dFactor(sin(x), cos(x)) :- !.
dFactor(cos(x), -sin(x)) :- !.
dFactor(sin(E), [cos(E)*DE]) :- dExpr(E,DE).
dFactor(cos(E), [-sin(E)*DE]) :- dExpr(E,DE).
dFactor(x^N, [N*x^(N1)]) :- N1 is N-1, !.
dFactor(E^N, [N*E^(Nm1)*(DE)]) :- dExpr(E,DE), Nm1 is N-1.

```

Tabella 6.6: Algoritmo tuProlog per la risoluzione simbolica delle derivate

6.2 Interfaccia Grafica

Tramite i tool di design di Visual Studio è stata realizzata una semplice interfaccia grafica che permette di:

- selezionare la teoria da testare;
- inserire la query tuProlog;
- avviare l'esecuzione.

- mostrare i risultati

Figura 6.7: Intefaccia utente della applicazione di collaudo

Infine sono state collegate:

- la selezione della teoria con il metodo handler `TheoryChanged` nel code behind. Si può notare come la creazione dell'oggetto `PrologDerivate` necessiti di un costrutto `try-catch` per gestire correttamente un eventuale eccezione dovuta ad un errato riferimento al file `der.pl`;

```
private void TheoryChanged(object sender, EventArgs e) {
    switch (((ComboBox)sender).SelectedItem.ToString()) {
        case "Permutazioni":
            prolog = new Perm();
            break;
        case "Derivate":
            try{
                prolog = new PrologDerivate();
            }catch{
                labelError.Text = "Cannot find prolog source";
            }
            break;
    }
}
```

Tabella 6.8: Gestione delle evento a seguito del cambio di teoria

- la pressione del tasto “GO” con il metodo handler `InputButton_Click`;

```
private void InputButton_Click(object sender, EventArgs e){
    OutputTextBox.Clear();
    try{
        if(prolog == null){
            labelError.Text = "Please Select Theory";
            return;
        }
        if (prolog is Perm){
            Term goal = new Struct(
                "perm", Term.createTerm(InputTextBox.Text), new Var("R")
            );
            SolveInfo SI = prolog.solve(goal);
            if(!SI.isSuccess())
                OutputTextBox.AppendText("no");
            while (SI.isSuccess()){
                OutputTextBox.AppendText(SI.getTerm("R").toString() + "\n");
                if(prolog.hasOpenAlternatives())
                    SI = prolog.solveNext();
                else
                    break;
            }
        }
        else
            if(prolog is PrologDerivate){
                Term goal = new Struct(
                    "dExpr", Term.createTerm(InputTextBox.Text), new Var("DT")
                );
                SolveInfo SI = prolog.solve(goal);
                if(!SI.isSuccess())
                    OutputTextBox.AppendText("no");
                else
                    OutputTextBox.AppendText(
                        SI.getTerm("DT").toString() + "\n"
                    );
            }
    }
    catch (MalformedGoalException ex) {
        OutputTextBox.AppendText("Malformed Goal\n");
    }
}
```

Tabella 6.9: interazione con il motore tuProlog

6.3 Osservazioni

Il test della applicazione di collaudo a dimostrato la piena funzionalità del generatore di codice che ha soddisfatto tutti requisiti preposti.

Capitolo 7

Conclusioni

In questa tesi ci si è posti l'obiettivo di rendere pienamente funzionante il framework P@J anche in tuProlog.NET, intendendosi con questo la possibilità di non soltanto di utilizzare in .NET codice P@J originariamente scritto in Java e tradotto via IKVM, ma di poter analogamente scrivere in Prolog l'implementazione di un metodo definito in un linguaggio .NET (ad esempio C#). Lo studio svolto nel capitolo 4 ha mostrato come, non sia possibile utilizzare le annotazioni del framework P@J come attributi .Net, poiché la tecnica di type erasure adottata dal compilatore Java rende impossibile recuperare i metodi che utilizzano tipi generici (come quelli sfruttati dal framework P@J) nel mondo .NET.

A causa di questo, il problema non è risolvibile senza modificare in profondità il modello di funzionamento stesso del framework P@J.

Poiché uno dei vincoli iniziali era appunto la non modifica dell'attuale design (e implementazione) di P@J, si è esplorata un diverso approccio, specificatamente pensato per il mondo .NET, basato sul concetto di generatore di codice, pienamente supportato dal tool "standard de facto" nel mondo .NET - Visual Studio.

Questo nuovo approccio si basa sull'idea di raggiungere lo stesso risultato – ovvero, consentire di scrivere in tuProlog l'implementazione di un metodo .NET – generando in modo automatico il codice di una classe .Net equivalente. Da includere poi normalmente del progetto (solution) dell'applicazione .Net.

In questo modo, non soltanto si è raggiunto il risultato voluto, ma si è ottenuta anche una maggiore separazione fra i linguaggi in quanto il codice tuProlog viene anche fisicamente posto in un file separato e integrato automaticamente nella classe .Net o a compile time o a runtime, a scelta del progettista. Ciò rende possibile continuare a sviluppare e collaudare le teorie tuProlog con gli usuali strumenti dedicati, senza dover aggiornare il codice .Net, a seguito di modifiche che non

incidano sulle teste delle clausole. Infatti cambiando la testa della clausola cambierebbe il punto di accesso da .Net verso tuProlog.

I template realizzati in questa tesi si riferiscono ai due più diffusi linguaggi della piattaforma .Net, ma il generatore di codice si presta già ad essere esteso per supportare gli altri linguaggi .Net. Al fine di estendere il supporto ad un altro linguaggio è sufficiente aggiungere due nuovi template, uno per ogni modalità di linking, e definire un nuovo metodo per gestire l'applicazione delle keyword all'interno dei template.

In definitiva, pur se con un approccio diverso da quello classico di P@J, l'obiettivo della tesi è stato completamente raggiunto. Questo approccio potrà essere rivisto ed eventualmente esteso/modificato in futuro, qualora si decida di rivedere in profondità l'approccio attualmente adottato nel framework P@J a livello di gestione dei tipi (e specificatamente della type inference) per valutare se, in una tale ottica, risulti ancora preferibile nel quadro .NET rispetto a una traduzione diretta, via IKVM, del corrispondente componente Java.

Bibliografia

1 tuProlog

<http://alice.unibo.it/xwiki/bin/view/Tuprolog/>

2 Java

<http://java.sun.com>

3 .NET framework

<http://www.microsoft.com/net>

4 Common Language Infrastructure (CLI)

<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf>

5 Mono .NET Framework

http://www.mono-project.com/Main_Page

6 DotGNU

<http://www.gnu.org/software/dotgnu/>

7 Common Language Runtime

<http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx>

8 Visual Basic

<http://msdn.microsoft.com/en-us/vstudio/hh388573>

9 Visual C++

<http://msdn.microsoft.com/en-us/vstudio/hh386302.aspx>

10 C#

<http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx>

11 Delphi

<http://www.delphibasics.co.uk/>

12 C++

<http://www.open-std.org/jtc1/sc22/wg21/>

13 Visual J#

[http://msdn.microsoft.com/en-us/library/7xsxf8e2\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/7xsxf8e2(v=vs.80).aspx)

14 Visual Studio

<http://www.microsoft.com/visualstudio/>

15 Visual Studio SDK

<http://msdn.microsoft.com/en-us/vstudio/vextend.aspx>

16 IKVM.NET

<http://www.ikvm.net/>