UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN SCIENZE DELL'INFORMAZIONE

PROGRAMMAZIONE LOGICA PER IL WEB SEMANTICO

Relatore: Prof. Alessandro PROVETTI
Correlatore: Prof.ssa Elisa BERTINO

Tesi di Laurea di:
Franco SALVETTI
Matr. Nr. 384576

ANNO ACCADEMICO 2001-2002

alla mia Mamma ed al mio Papà

Indice

Indice iii

Elenco delle Figure vi

Abstract vii

Ringraziamenti viii

Introduzione 1

1 Gli ideali del Web semantico 5

1.1 Il Web . . . 5
1.1.1 La nascita del Web . . . 6
1.1.2 Il Web oggi . . . 6
1.1.3 Il Web del futuro, il Web semantico . . . 6
1.1.4 Obiettivi del Web semantico . . . 9
1.1.5 I servizi . . . 10
1.2 Introduzione al Web semantico . . . 14
1.2.1 Il principio del 'Least Power' . . . 14
1.2.2 La rappresentazione della conoscenza . . . 15
1.2.3 I nomi delle risorse: URI . . . 15
1.2.4 Lo strato del mark-up: XML . . . 16
1.2.5 Lo strato dei modelli base: RDF . . . 16
1.2.6 Lo strato degli schemi: RDFS . . . 17
1.2.7 Lo strato delle ontologie . . . 18
1.2.8 Lo strato delle conversioni: DAML+OIL . . . 19
1.2.9 Lo strato logico . . . 20
1.2.10 Lo strato della Fiducia: Web of trust . . . 20
1.3 La rappresentazione della conoscenza . . . 22

1.3.1 Il Web visto come una base conoscenza 23
1.3.2 Le reti semantiche 25
1.4 Il problema dei nomi 29
1.4.1 Le URI 29
1.5 Il mark-up 32
1.5.1 Tipi di mark-up 33
1.5.2 Visualizzazione del mark-up 34
1.5.3 Le componenti del mark-up 35
1.6 I linguaggi di mark-up 37
1.6.1 HTML 38
1.6.2 XML 39
1.7 Resource Description Framework (RDF) 53
1.7.1 Il modello RDF 54
1.7.2 Rappresentazioni equivalenti del modello RDF 60
1.7.3 Sintassi RDF XML 62
1.7.4 Sintassi abbreviata RDF XML 68
1.7.5 I contenitori 70
1.7.6 Asserzioni su asserzioni 74
1.8 RDF Schema Language 75
1.8.1 Le Classi base di RDFS 79
1.8.2 Le Proprietà base di RDFS 85
1.8.3 Esempio di Schema RDFS 89
1.8.4 Gli Schemi di RDF ed RDFS 93

2 DAML+OIL 94
2.1 Introduzione a DAML+OIL 94
2.2 Perché DAML+OIL? 96
2.3 L'ontologia DAML+OIL 97
2.4 DAML-S 102
2.5 DAML-L 103

3 Programmazione logica: Answer Set Programming 105
3.1 Modelli stabili 105
3.1.1 Sintassi 105
3.1.2 Semantica 106
3.1.3 Answer Set Programming 108
3.2 Alcuni esempi 109
3.3 Implementazioni esistenti 112
3.3.1 L'espressività di ASP 113

4 Semantica esplicita per DAML+OIL **115**
4.1 Introduzione 115
4.2 La rete semantica di partenza 118
4.3 Da rappresentazione a conoscenza 118
4.4 Preliminari 120
4.5 La semantica esplicita 122
4.6 La Validazione 134

5 Sperimentazione **150**
5.1 Introduzione 150
5.2 Le nuove conoscenze 150
5.3 Esempi 152

6 Estensioni per DAML+OIL **162**
6.1 Introduzione 162
6.2 I default 163
6.3 Pingu 165

Conclusioni **175**

Appendice A **177**

Appendice B **178**

Appendice C **179**

Appendice D **180**

Appendice E **181**

Appendice F **185**

Appendice G **195**

Appendice H **197**

Glossario **198**

Risorse **200**

Bibliografia **203**

Elenco delle figure

1.1 Un esempio di rete semantica [esempio: 1.3.1]. 29
1.2 *La tesi sul Web semantico ha come autore Franco*. 57
1.3 *Franco* ora è una risorsa. 58
1.4 Una semplice asserzione RDF [esempio: 1.7.8]. 65
1.5 Un contenitore RDF [esempio: 1.7.13]. 72
1.6 Descrizione delle risorse contenute in un `rdf:Bag` [esempio: 1.7.14]. . 74
1.7 Asserzioni su asserzioni in RDF [esempio: 1.7.15]. 76
1.8 Grafo delle relazioni tra le classi e le proprietà base di RDFS. 81
1.9 `Veicolo` è una classe [esempio: 1.8.1]. 83
1.10 Classi e sottoclassi nel dominio semantico dei *veicoli* [esempio: 1.8.10]. 92

5.1 Rete semantica prima dell'inferenza [esempio: 5.3.1]. 153
5.2 Rete semantica dopo l'inferenza [esempio: 5.3.1]. 156
5.3 Definizione circolare delle sottoclassi [esempio: 5.3.5]. 160

6.1 `pingu` non vola e `magic` vola. 167
6.2 Ci sono due modelli alternativi; in uno `pingu` vola mentre nell'altro no. 172
6.3 `pingu` può volare perchè eredita di essere un `Uccello`. 174

Abstract

DAML+OIL è stato definito per facilitare l'interoperabilità semantica. Non avendo in sé meccanismi inferenziali ma solo costrutti linguistici per descrivere le nostre conoscenze sul *mondo* è necessario, al fine di effettuare dell'inferenza, fornire una nozione di *verità* – ossia dare una teoria dei modelli. Per fare ciò in questa tesi *tradurremo* le asserzioni primitive del linguaggio DAML+OIL in un programma logico π_1 scritto per SMODELS, *esplicitandone* così la semantica. In questo modo, dato un programma logico π_2 costituito da asserzioni DAML+OIL avremo che i modelli stabili di $\pi_1 \cup \pi_2$ saranno gli insiemi massimamente consistenti di fatti veri inferibili dalle asserzioni stesse, rispetto alla semantica da noi definita.

Introdurremo il concetto di località per asserzioni DAML+OIL e definiremo una semantica alternativa che permetta di *ragionare* in termini di *mondi chiusi locali*. Fatto ciò sarà possibile introdurre nella semantica di DAML+OIL la teoria dei *default* e la *negazione per fallimento*. In prospettiva ciò permetterà di utilizzare all'interno del Web semantico meccanismi di ragionamento per *senso comune*.

Una semantica esplicitata attraverso dei programmi logici fornirà altresì semplici meccanismi di validazione rispetto alle struttura base di DAML+OIL, e suggerirà l'idea di come sia possibile dare la semantica di classi e proprietà utente direttamente nella loro definizione.

Ringraziamenti

Voglio ringraziare Alessandro Provetti, il mio relatore, per i suoi continui suggerimenti e per il suo costante supporto durante questo lavoro di ricerca. Voglio inoltre ringraziare Massimo Santini per la sua guida durante i miei primi anni di Università.

Ovviamente sono grato ai miei genitori per la loro grande pazienza ed *amore*. Senza di loro questa tesi non sarebbe mai stata scritta (letteralmente).

Voglio inoltre ringraziare: Alan Waugh (perché uno *statement* può aprire molte porte); Alessandra Pavani (che mi ha sempre ascoltato in tutti questi anni); Antonella Scoliero (per avermi fatto realizzare il sogno di girare un video); Betty (per avermi chiesto cosa aspettavo a finire); Bobby (perché alla fine non abbiamo poi tanti veri amici); Carla Browne (per avermi dedicato tante ore durante la preparazione al TOEFL); Gianni Tomasi (per essermi sempre stato amico); Gigi (per i *cappelli* di Shauder); Giovanni (per essere stata una persona amica ad Oxford); Jacopo Mantovani (per avermi aiutato e *tenuto a bada* nell'ultimo anno); Marco Bassi (*mens sana in corpore sano*); Mario Borroi (perché gli esempi servono più di mille parole); Mario Mazzeo (per avermi fatto sentire regista per un giorno); Mark Jennings (per la sua amicizia durante il periodo di studio ad Oxford); Pinuccia Rubini (per il suo continuo ed efficace supporto morale); Prof. Bellettini, Prof. Ghilardi, Prof.ssa Pagnoni, Prof. Mundici (per aver creduto nelle mie capacità aiutandomi nel difficile processo di ammissione al Master negli Stati Uniti); Roberta Gambini (per avermi detto di No); Roberto Radicioni ed Alessandra Mileo (per il lavoro fatto su BRACCOBALDO); Saint Germain (per avermi fatto alzare tutte le mattine del 2001); Salvatore (un occhio vigile può cambiare la vita); Valentina Matteazzi (per avermi aiutato in tutto quello che poteva); ed infine Franco Salvetti (per averci sempre creduto e non aver mai mollato in tutti questi ultimi difficilissimi dieci anni).

Milano, Italy — Franco Salvetti
27 giugno 2002

Introduzione

L'idea di Web semantico è stata introdotta da Tim Berners-Lee, l'inventore del protocollo `http`, come estensione dell'attuale Web. L'obiettivo del Web semantico è di aumentare l'usabilità del Web attraverso la marcatura di una pagina, in modo da permettere ad *agenti software* di operare in modo autonomo ed efficace.

Il problema è in prima battuta un problema di rappresentazione della conoscenza. I sistemi tradizionali di rappresentazione della conoscenza fanno affidamento sul fatto che la definizione di un concetto è condivisa a priori. Questo sul Web non è pensabile, ma vorremmo comunque integrare e condividere ogni sorgente di informazione.

> "*The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation*". – Tim Berners-Lee, James Hendler, Ora Lassila

La maggior parte del contenuto del Web attuale è progettata per essere letta da esseri umani e non per essere manipolata in modo automatico da programmi. Inoltre il ruolo del Web sta evolvendo: da un fornitore di documenti ed immagini ad un fornitore di servizi. I computer possono analizzare una pagina Web alla ricerca di marcature (questo è un titolo, questa è una immagine, questo è un link), ma non hanno un sistema adeguato per trattarne la semantica (questa è la home-page del DSI, questo

link porta agli orari di ricevimento del Prof. Verdi o questa è la pagina con i requisiti per accedere al Master in Comunicazione digitale ecc.) in modo assolutamente indipendente dalla veste grafica o di presentazione.

Per realizzare il Web semantico ci si appoggia su due importanti tecnologie: l'eXtensible Markup Language (XML) e il Resource Description Framework (RDF). Il linguaggio XML permette sostanzialmente di definire dei *tag* (etichette che commentano porzioni di pagina) e di dotare di struttura le pagine Web. L'uso di questi tag può essere anche molto sofisticato *ma non permette di esplicitarne una semantica.* Il significato può venire espresso con RDF, che lo codifica attraverso delle asserzioni fatte da delle triple della forma soggetto-attributo-valore. Soggetto, attributo e valore sono identificati da un Universal Resource Identifier (URI) di cui la URL è l'esempio più comune. Attraverso un meccanismo ben definito, chiunque può costruire una URI per identificare un soggetto (e.g. Franco Salvetti) un attributo (e.g. autore) un valore (e.g. tesi sul Web semantico). Questo sistema di triple sparse su pagine diverse all'interno del Web forma un grafo che dà l'interconnessione dell'informazione codificata. L'utilizzare le URI fa sì che l'informazione non sia una semplice stringa di caratteri, ma sia sempre un link alla sua precisa definizione.

Un altro componente fondamentale del Web semantico è RDF Schema (RDFS) che, mettendo a disposizione una minima struttura di classi e proprietà nonché una struttura gerarchica tra esse (e.g. attraverso la proprietà `subClassOf`), fornisce lo strumento linguistico per la loro creazione. Raccolte di nuove classi e di nuove proprietà, reperibili sul Web ad una data URL e descritte anch'esse attraverso delle triple, prendono il nome di *ontologie*. Chiunque può definire o usare delle ontologie creando istanze di classi e predicando su di esse.

Vista l'eterogeneità e la decentralizzazione del Web sono altresì necessari meccanismi di condivisione di ontologie. Dovrà infatti essere possibile asserire che due nomi diversi, definiti in due ontologie diverse, afferiscono allo stesso oggetto. A questo scopo è stata definita l'ontologia DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer), che definisce proprietà e classi generali che permettono l'interoperabilità, nonché il raffinamento, di ontologie liberamente pubblicate sul Web. In questo modo sarà possibile facilitare la programmazione di applicazioni che accedano autonomamente al Web con un comportamento adattivamente rispetto alle modifiche dei contenuti.

Quello che manca è una definizione accettata di quali operazioni (inferenze) siano possibili (valide) su un insieme di asserzioni distribuite su più pagine Web. Non è infatti ancora stato dato e recepito uno schema di inferenza corretto, nel senso logico del temine, con il quale sia possibile progettare ed implementare un sistema di ricerca o navigazione automatico del Web semantico il più possibile autonomo dall'utente e che tenga conto delle problematiche dell'inferenza detta per *default*, dell'assunzione di *mondo chiuso* e della *negazione*. I default, sotto l'assunzione di mondo chiuso locale, permettono ad un sistema deduttivo automatico di realizzare dell'inferenza in situazioni di mancanza di informazione esplicita contraria.

In questa tesi abbiamo dato una risposta originale a questo problema, molto sentito all'interno della comunità di sviluppo del Web semantico. Proponiamo infatti una semantica dichiarativa delle asserzioni base dell'ontologia DAML+OIL ed un meccanismo inferenziale, implementato e sperimentabile su insiemi di asserzioni di piccola taglia. La semantica è data per traduzione delle asserzioni DAML+OIL verso l'Answer Set Programming (ASP). L'ASP è definito, in breve, dalla sintassi DATALOG

per le basi di dati deduttive, più la negazione per fallimento, la cui semantica è stata definita da Gelfond e Lifshitz [5].

Il punto-chiave del nostro lavoro è che i predicati DAML+OIL definiti dal consorzio DAML vengono tradotti in un programma logico che usa gli argomenti dell'asserzione per concludere nuove istanze del predicato stesso. Il risultato è che inferenze basate su *default inheritance*, tassonomie e altri aspetti del ragionamento su insiemi di oggetti vengono catturati in modo chiaro e diretto dagli answer set del programma. I motori inferenziali per ASP permettono di automatizzare questi tipi base di ragionamento e di verificare che un insieme di asserzioni DAML+OIL sia consistente con la sua definizione nonché di completare una definizione utente attraverso l'applicazione della semantica delle proprietà DAML+OIL da noi esplicitata.

La tesi è organizzata come segue: nel primo capitolo consideriamo gli ideali del Web semantico e delineiamo la situazione attuale dei linguaggi per asserzioni ed ontologie. Il secondo capitolo è dedicato a DAML+OIL e a motivare la nostra scelta di tale linguaggio, anche in vista delle estensioni a DAML for Services (DAML-S) e DAML Logic (DAML-L). Nel terzo capitolo viene presentato l'Answer Set Programming [9] (ASP), le sue definizioni, ed il risolutore SMODELS usato per la parte implementativa della tesi. Il quarto capitolo presenta la traduzione dell'insieme base dalle asserzioni DAML+OIL ad ASP dove mostriamo il risultato per asserzioni che usano le proprietà insiemistiche rilevanti. Il quinto capitolo è dedicato all'analisi della sperimentazione con ontologie già disponibili sul Web. Infine, descriviamo due estensioni delle definizioni DAML+OIL che ci apprestiamo a sottoporre al consorzio DAML.

Capitolo 1

Gli ideali del Web semantico

In questo capitolo consideriamo gli ideali del Web semantico e delineiamo la situazione attuale dei linguaggi per asserzioni ed ontologie. Partendo dagli scopi che si prefigge il Web semantico mostreremo in dettaglio la sua struttura a *livelli*. Vedremo i meccanismi per nominare individui del dominio materiale, l'uso dei linguaggi di mark-up e delle reti semantiche per la descrizione delle nostre conoscenze sul mondo. Per arrivare poi ai due linguaggi cardine del Web semantico, il Resource Description Framework (RDF) e l'RDF Schema Language (RDFS) e con essi le ontologie.

1.1 Il Web

Il WWW (World Wide Web), più brevemente il Web, è un servizio Internet che può essere visto come ciò che permette l'accesso alla raccolta di milioni di documenti collegati tra loro e dislocati su computer residenti in tutto il mondo. Internet è il mezzo per mettere in contatto milioni di computer mentre il Web è un sistema di comunicazione che permette di trasferire gli *ipermedia*. In definitiva, il Web non è altro che un sistema hardware e software che utilizza Internet come ponte di collegamento

per la comunicazione e la condivisione.

1.1.1 La nascita del Web

Il Web nasce nel Marzo del 1989. La sua scoperta è dipesa dall'esigenza dei ricercatori del CERN (Centro Europeo per la Ricerca Nucleare) sparsi per il mondo di condividere informazioni e scoperte scientifiche in tempo reale. Tim Berners-Lee ideò il Web come sistema ipertestuale di comunicazione tra computer. I ricercatori crearono così una rete di documenti collegati tra loro. Il Web cominciò a essere utilizzato nel 1992; prevedeva la registrazione dei documenti su computer chiamati *Web server* e la consultazione di questi documenti utilizzando software particolari in grado di leggere le pagine ipertestuali.

1.1.2 Il Web oggi

Tra i vari servizi Internet (FTP - File Transfer Protocol, telnet, posta elettronica ecc.) quello che ha raggiunto il più elevato livello di diffusione e sviluppo è sicuramente il Web. Esso consente all'utente di muoversi (navigare) attraverso un numero illimitato di documenti collegati. Il WWW aiuta l'utente nel reperimento delle informazioni e delle risorse guidandolo attraverso una rete di hyperlink.

1.1.3 Il Web del futuro, il Web semantico

Con Web semantico ci si riferisce al nome, o per meglio dire, alla visione del Web del futuro che ha il W3C (World Wide Web Consortium) ed in particolare il suo uomo simbolo: Tim Berners-Lee (l'inventore di WWW, URI - Universal Resource Identifier, http - hyper text transfer protocol e HTML - HyperText Mark-up Language). È

infatti Tim Berners-Lee che, dopo il trasferimento del W3C dal CERN di Ginevra ai laboratori dell'MIT (Massachusetts Institute of Technology) di Boston, ha lanciato l'idea del Web del futuro. Un Web semantico in cui le informazioni siano *machine understandable*.

L'informazione presente sul Web attuale è processabile dalle macchine per scopi di visualizzazione, grazie all'HTML, o sintattici, attraverso l'XML (eXtensible Mark-up Language), ma non *semantici*. I motori di ricerca sono in grado di trovare informazioni relative a Giuseppe Garibaldi, per esempio, ma non sono in grado di distinguere quando la stringa `Giuseppe Garibaldi` si riferisce al personaggio storico, ad una pizzeria, ad una via o ad un proprio vicino di casa che si chiama Giuseppe Garibaldi. Per superare questo problema si deve *fornire* di semantica il contenuto del Web.

La prima fase di progetto del Web semantico prevede quindi di realizzare un modello comune di rappresentazione dell'informazione che sia condiviso da tutti, che sia sufficientemente espressivo, porti con sé un *valore* semantico e cosa più importante sia comprensibile dalle macchine.

La possibilità di rendere disponibili sul Web dati che siano *comprensibili* da agenti software sta diventando un'alta priorità per molte comunità a livello mondiale. Il Web potrà raggiungere le sue piene potenzialità solo se diventerà un *luogo* dove i dati potranno essere condivisi ed elaborati tanto bene da entità software quanto da esseri umani. Per le dimensioni del Web, i programmi di domani dovranno essere disegnati per condividere ed elaborare dati anche quando i programmi saranno stati scritti in modo completamente indipendente dal contesto applicativo nel quale si troveranno ad operare.

Le informazioni, al posto di essere memorizzate in testi scritti in linguaggio quasi

naturale [HTML], saranno mantenute in una forma strutturata che permetta di essere acceduta sia dai computer che dalle persone. Tutte le conoscenze saranno espresse attraverso delle asserzioni descrittive che *dicono* quale relazione sussiste tra due entità (e.g. *Pia è la mamma di Franco*). Un enorme quantità di conoscenze può essere espressa in questo modo *Un certo libro costa 16€*, *Franco ha come città di residenza Milano*, *La Terra ha come satellite la Luna*.

Il progetto di Tim Berners-Lee non è ne un progetto facile, né un progetto privo di detrattori. Legato al concetto di Web semantico vi è quello di fornitura di servizi attraverso il Web, tema che sta molto a cuore all'industria, tanto che imputando al W3C una eccessiva lentezza nel fornire delle linee guida ha fondato separatamente il WS-I: una cordata di industrie private che raccoglie cento tra le più importanti realtà informatiche a livello mondiale con lo scopo di sviluppare le linee guida per la realizzazione di servizi sul Web. Anche se sulla carta il WS-I dovrebbe collaborare con il W3C, i dubbi sulle loro divergenze sembrano a tutt'oggi molto fondati.

Le critiche però non arrivano solo dal fronte dell'industria, ma anche da quello accademico.

> "*The other thing is that the logicians, in spite of their sometimes exasperatingly mathematical writing style, don't spend most of their time counting angels. They have done a lot of extremely hard conceptual work for us already, and it just seems damn silly not to try to take advantage of it. In particular, if they say that some parts of the beach are slippery, it seems foolhardy to choose that very spot to erect our new headquarters*".

In questo passaggio estratto da una e-mail scritta da Pat Hayes, una delle persone più autorevoli a livello mondiale nell'ambito dell'Intelligenza Artificiale, è sintetizzato

il pensiero che molto lavoro è già stato fatto dai logici in questi anni, lavoro al quale si dovrebbe attingere per sviluppare il Web semantico.

1.1.4 Obiettivi del Web semantico

Vari sono gli obiettivi che si prefigge il progetto del Web semantico, ecco in sintesi quelli più importanti e qualificanti:

- facilitare l'automatizzazione dell'interazione tra agenti software ed informazioni;
- accrescere il livello di fiducia dell'utente nella *bontà* delle risposte ottenute dal Web semantico;
- aumentare l'accessibilità dell'informazione;
- permettere l'automazione di transazioni di tipo commerciale;
- fornire gli strumenti necessari al riuso dell'informazione;
- facilitare l'integrazione di informazioni già presenti sul Web semantico;
- aumentare l'efficienza dei motori di ricerca;
- realizzare dei sistemi di catalogazione dei contenuti e delle relazioni tra le pagine;
- creare delle collezioni (ossia degli insiemi) di pagine che, pur appartenendo a domini diversi siano correlino le varie informazioni da un punto di vista semantico;
- gestire i diritti d'autore e la proprietà intellettuale delle opere dell'ingegno.

1.1.5 I servizi

Una problematica strettamente connessa con il Web semantico è quella dei Servizi Web. Il Web in futuro non sarà più un mero deposito di testo ed immagini, ma un fornitore di servizi. Il cambio di paradigma sarà nel passaggio dall'accesso all'informazione alla richiesta di un servizio che fornisce informazione e più in generale di un servizio che realizza una transazione, ossia uno scambio di informazione; il tutto in modo trasparente per l'utente.

Definizione 1.1.1. Per *servizio di informazione* intendiamo un servizio fondamentalmente unidirezionale in cui un utente richiede delle informazioni.

Esempio 1.1.1. Un servizio che risponda alla richiesta *In qualità di studente straniero, qual è il minimo punteggio TOEFL che si deve avere per l'ammissione ad un Master in CS presso il Dipartimento di Computer Science della Stanford University*, può essere inteso come un *servizio di informazione.*

Definizione 1.1.2. Per *servizio di transazione* intendiamo un servizio di tipo transazionale, un servizio in cui cioè verrà realizzata una transizione tra l'*agente* dell'utente e l'*agente* del fornitore del servizio per ottenere un preciso scopo.

Esempio 1.1.2. Un servizio che svolgesse la richiesta dell'utente *compila la domanda di ammissione al corso di MS in Computer Science che inizia nell'autunno del 2002 presso la Stanford University* può essere inteso come un servizio di transazione. Risulta evidente che in questo caso vi sarà una iniziale richiesta del servizio da parte dell'utente che si tradurrà in una successiva serie di richieste di informazioni da parte dell'agente del fornitore del servizio, al fine di ottenere gli scopi desiderati dall'agente dell'utente che lavora in vece dell'utente stesso.

Facendo alcune semplificazioni possiamo pensare che, oggi come oggi, le informazioni sul Web siano contenute all'interno di pagine HTML. Ricordiamo in breve che l'HTML è un linguaggio che permette di descrivere come visualizzare delle informazioni, tipicamente delle stringhe di testo. Quindi, tolta la parte di presentazione, una pagina HTML è semplicemente testo, collegamenti ipertestuali e immagini.

Per poter fornire dei servizi che non prevedano l'interazione diretta dell'utente con la pagina stessa si possono scrivere dei programmi, ad esempio in C++, che partendo da alcune conoscenza preacquisite sulla pagina stessa attraverso una serie di elaborazioni estraggano da essa l'informazione voluta.

Pensiamo al caso del sito del Computer Science dept. della Stanford University e pensiamo di voler conoscere il punteggio TOEFL minimo richiesto per poter fare domanda di ammissione. Si può pensare di scrivere un programma C++ che, partendo dalla home-page del dipartimento ed *inseguendo* i link presenti nelle varie pagine ad essa collegate, trovi pagine che contengano la stringa TOEFL. A questo punto una ricerca di stringhe basata ad esempio sul concetto di località potrebbe trovare il punteggio che si sta effettivamente cercando. Sempre con un alto livello di semplificazione possiamo pensare che questo sia l'atteggiamento degli attuali motori di ricerca che fondamentalmente si basano su operazioni di string matching.

Tornando all'esempio precedente, possiamo facilmente intuire che la quantità di casi in cui la nostra ricerca porti a risultati sbagliati o incompleti sia molto alta. Come possiamo quindi pensare di fornire un semplice servizio di informazioni affidabile? Perché è così difficile *capire* quale sia il punteggio TOEFL minimo richiesto per accedere alla Stanford University?

Il problema è legato al fatto che HTML serve per descrivere la visualizzazione

dell'informazione che deve poi essere fruita direttamente da un agente umano il quale è dotato di una sua *intelligenza* che gli permette di capire il significato di ciò che sta vedendo.

Un essere umano non avrà problemi per poter evincere l'informazione cercata all'interno della frase - *Il punteggio TOEFL minimo richiesto nell'anno 2002 è 230.* Di certo non verrà imbrogliato da quel 2002 e capirà immediatamente che 230 è l'informazione cercata. In qualche modo si inizia ad intuire che quello che serve per pensare di automatizzare questo servizio è un agente in grado di esibire lo stesso comportamento intelligente esibito dal nostro agente umano.

Il problema che si evidenzia è già sufficientemente complicato a livello di servizio di fornitura di informazioni. È quindi facile intuire che la cosa si complichi notevolmente nel momento in cui si passi al livello in cui è richiesto un servizio che prevede l'interazione tra il nostro agente intelligente ed il sito della Stanford University nel caso ad esempio della compilazione della domanda effettiva di iscrizione.

Tornando per il momento al caso *semplice*, quello che abbiamo detto è che potremmo pensare di realizzare un agente intelligente. Ma come? Possiamo trovare il modo di *aiutare* l'agente intelligente? Quello che si può facilmente intuire è che un testo è privo di semantica in sé e che la semantica è una sua qualità attuale in presenza di un lettore. Potremmo però pensare ad un bambino che vogliamo aiutare nella lettura di un testo complicato. Potremmo ad esempio dire al bambino che dovesse leggere la frase - *Il punteggio TOEFL minimo richiesto nell'anno 2002 è 230* - che TOEFL è un esame che qualsiasi studente straniero deve sostenere per accedere ad una università americana e che 230 è il punteggio minimo richiesto per l'anno 2002. Forse il bambino non avrà ancora capito ma almeno ci avremo provato. Potremmo quindi pensare di

rifrasare quanto detto in modo più strutturato *marcando* le parti costituenti la frase con delle stringhe che ne facilitino la comprensione:

```
[inizio dei requisiti]
  [un numero]{Il punteggio}
  [esame di inglese]{TOEFL}
  [strettamente]{minimo}
  [non è possibile accedevi senza]{richiesto}
  [i corsi iniziano ai primi di settembre]{nell'anno 2002}
  [collegamento semantico a {Il punteggio}]{è}
  [valore numerico per i Computer Based Test]{230}
  [terminatore della frase]{.}
  :
  :
[fine dei requisiti]
```

In questo modo se la semantica di delle parti espresse tra parentesi quadre fosse ben nota avremmo un modo forse più automatico di interpretare il significato di quanto appare sullo schermo. Sfruttando ciò potremmo quindi pensare ad un agente intelligente che una volta *arrivato* sulla pagina in questione sia in grado di evincere da essa che tra i requisiti di ammissione a Stanford vi è il TOEFL con punteggio minimo richiesto 230.

Tutto ciò apre una serie di domande: Come siamo arrivati sulla pagina in questione? Come possiamo definire un linguaggio che permetta di definire la semantica di un testo? Come deve essere *impacchettata* la richiesta di servizio da parte dell'utente? A chi deve essere inviata la richiesta di servizio? Domande alle quali il Web semantico vorrebbe dare una risposta.

A questo punto si può cominciare ad intuire che la comprensione del testo sopra citato nella sua versione *marcata* dipende dalla capacità di un agente di conoscere il significato dei *marcatori*. È quindi chiaro che l'idea che verrà sviluppata è quella

di realizzare dei meccanismi per la *costruzione* di marcatori al fine di condividerne il significato.

1.2 Introduzione al Web semantico

Il Web semantico non è un oggetto monolitico ma costruito attraverso *strati* sovrapposti di *linguaggi*. Sono infatti molte le parti che permetteranno di raggiungere i vari obiettivi che si prefigge il Web semantico. Ogni nuovo strato usa o estende gli strati precedenti. Vediamo ora la *stratificazione* proposta dal W3C.

1.2.1 Il principio del 'Least Power'

Un principio base che permetterà di capire una sorta di meccanismo a scatole cinesi in cui sono definiti i linguaggi che serviranno per descrivere i contenuti del Web semantico è quello del Least Power che più o meno si può esprime come:

> "*Quando esprimi qualche cosa, usa sempre il linguaggio meno espressivo possibile*".

Una sorta di *versione informatica* del più noto *Rasoio di Ockham*:

> "*Entia non sunt multiplicanda praeter necessitatem*". – G. da Ockham[1]

Si tratta di un principio metodologico che sta alla base del pensiero scientifico moderno: tra due teorie entrambe capaci di spiegare un gruppo di dati occorre scegliere quella più semplice e dotata di un minor numero di ipotesi, *tagliando via* con il rasoio

[1] Guglielmo da Ockham, francescano, filosofo e teologo inglese (1295-1350), sostenne che, per intendere la realtà, non bisogna postulare enti inutili, ma attenersi a quelli necessari (rasoio di Ockham).

di Ockham quella più lunga e involuta. Non tanto perché una sia più *vera* dell'altra, quanto perché quella più breve e compatta permette di risparmiare tempo e fatica inutili.

Nel nostro caso l'idea è quella di avere un linguaggio semplice e *snello* che però abbia una espressività sufficientemente alta da permettere di esprimere tutti i contenuti del Web semantico.

1.2.2 La rappresentazione della conoscenza

I vari problemi connessi con il Web semantico sono in prima battuta problemi di rappresentazione della conoscenza. Se in un database locale il significato di ogni singolo campo è ben noto a tutti, quello che si sta prefigurando per il Web semantico è una situazione nella quale ognuno possa definire i propri campi con un proprio significato. Si sollevano quindi due problemi, il primo è quello del significato dei campi, mentre il secondo è quello della struttura stessa dell'informazione. L'idea del Web semantico è di realizzare un modello base comune per tutti, che permetta la strutturazione dell'informazione e al contempo l'esplicitazione della semantica.

1.2.3 I nomi delle risorse: URI

Per poter descrivere le conoscenze che abbiamo sul mondo si devono poter nominare gli oggetti in esso contenuti, siano essi materiali (e.g. la bottiglia vuota che è sul mio tavolo) o immateriali (e.g. la relazione di parentela tra genitore e figlio). All'interno del Web semantico l'idea è di utilizzare come nomi delle URI (Universal Resource Identifier) per ogni oggetto di cui si voglia parlare. Essendo le URL (Universal Resource Locator) delle particolari URI, vedremo in seguito vari modi per poter associare

un oggetto ad una URL. Definiremo poi *risorsa* una qualsiasi cosa che sia associabile ad una URI.

1.2.4 Lo strato del mark-up: XML

Abbiamo accennato al problema della rappresentazione della conoscenza sul Web semantico. L'approccio base che si segue è di marcare dei testi per fornirli contemporaneamente di significato e di struttura. I meccanismi di mark-up utilizzati per il Web semantico sono riconducibili ad XML (eXtensible Mark-up Language) che vedremo in seguito, anche se prescindono da esso.

Definizione 1.2.1. Un *linguaggio di mark-up* è un qualsiasi linguaggio che renda esplicita l'interpretazione di una parte di testo.

1.2.5 Lo strato dei modelli base: RDF

Come vedremo questo modello comune di rappresentazione della conoscenza è il linguaggio RDF (Resource Description Framework); almeno, per quanto riguarda le aspettative e le ricerche che si stanno effettuando presso il W3C.

L'essere generale e semplice è la prerogativa di RDF il quale definisce in sé solo il significato di asserzione e di citazione (asserzioni su asserzioni). Questa semplicità prevede quindi la costruzione di strati successivi che si appoggino sulla possibilità di effettuare asserzioni e citazioni. C'è da dire però che le asserzioni di RDF potranno essere fatte non solo sui dati ma anche su metadati, verranno cioè fatte asserzioni su oggetti che sono a loro volta elementi che serviranno per descrivere i dati.

Nel linguaggio non sono definiti né il concetto di negazione né quello di implicazione, rendendolo perciò estremamente limitato; si potranno fare solo delle asserzioni su delle risorse. D'altro canto, dato un insieme di fatti, dare una dimostrazione per ogni query effettuata diventa estremamente semplice, dato che né i fatti né le query possono avere una struttura espressiva eccessivamente complicata.

Sebbene esistano delle applicazioni di RDF a livello di pura rappresentazione di dati [Dublin Core], [P3P] e [PICS] quello a cui noi siamo interessati è la possibilità di realizzare una linguaggio comune per il Web sul quale però sia possibile *montare* dei motori inferenziali.

L'idea è quella di creare dei meccanismi di interrogazione molto potenti che agiscano sul Web nella sua interezza e che si riferiscano a dati che individualmente siano espressi in un linguaggio molto semplice e non troppo espressivo, RDF appunto.

Il modello base di RDF, come vedremo in seguito attraverso degli esempi, permette di fare asserzioni o citazioni, permettendo quindi di mappare le nostre conoscenze sul mondo con questo nuovo linguaggio.

RDF sarà definito attraverso una propria grammatica EBNF (Extended Backus-Naur Form) ma sarà possibile esprimere asserzioni RDF attraverso un altro linguaggio di uso generale che è appunto XML.

1.2.6 Lo strato degli schemi: RDFS

Al fine di aumentare le capacità espressive del modello base viene introdotto uno strato che prende il nome di Schema. Questo strato fornisce un insieme minimo di classi e proprietà che permetteranno di *creare* nuove classi e nuove proprietà utente.

Sulle nuove proprietà che serviranno per descrivere il mondo vorremmo però poter

esprime dei vincoli in modo tale da dire per esempio quali tipi di risorse potranno essere descritte da questa nuova proprietà. Questo meccanismo permetterà di realizzare dei rudimentali strumenti di validazione di un documento. Sarà possibile ad esempio verificare che la proprietà `autore` descriva risorse che siano nomi di persone e non per esempio titoli di libri.

Vedremo in seguito il linguaggio RDF Schema che servirà appunto per esprimere nuove proprietà. Non è però detto che questo sia l'unico linguaggio possibile per definire nuove proprietà da utilizzarsi all'interno di un documento RDF. Si potrà infatti pensare di rendere più potente questo strato introducendo una parte logica che permetta di esprimere più complicate strutture di vincoli sulle nuove proprietà che si andranno via via ad introdurre.

1.2.7 Lo strato delle ontologie

Gli strumenti linguistici messi a disposizione dallo strato degli Schemi sono sufficienti per la creazioni di *collezioni* di proprietà e classi organizzate in modo gerarchico tra loro. Vedremo come questi vocabolari saranno le *ontologie* di riferimento per il Web semantico.

Definizione 1.2.2. Nell'ambito dell'AI (Artificial Intelligence), per *Ontologia* si può intendere l'atto iniziale e originale di distinzione degli individui (componenti base) dallo sfondo indistinto del dominio del discorso.

Commento 1.2.1. Per la spiegazione dei componenti dell'ontologia, ci si affida a frasi in linguaggio naturale e, in ultima analisi, all'intuizione. Inoltre, gli elementi dell'ontologia possono essere predicati, ma non ulteriormente dettagliati; sono cioè atomici.

Esempio 1.2.1. Se descriviamo un dominio dinamico mediate un'Ontologia temporale di secondi, non possiamo descrivere eventi/mutazioni di durata inferiore all'unità, né tantomeno possiamo usare tale descrizione per dettagliare cosa sia un secondo (mentre possiamo benissimo definire l'ora, il giorno ecc.)

Definizione 1.2.3. Nell'ambito dell'AI a volte per *ontologia* si intende una classificazione tassonomica di tali componenti base (gli individui) in classi e sottoclassi.

Esempio 1.2.2. La classificazione linneiana[2] delle piante è una *ontologia*.

Nell'ambito del Web semantico, Tim Berners-Lee [1] ha dato questa definizione:

> "*Una ontologia è una collezione di frasi che definiscono le relazioni tra concetti e specificano le regole logiche*".

1.2.8 Lo strato delle conversioni: DAML+OIL

Abbiamo visto che uno Schema servirà per descrivere delle nuove proprietà. Sarà quindi possibile che in due *ontologie* diverse sia stata descritta una proprietà con due nomi diversi ma con lo stesso significato. Sarà quindi necessario un meccanismo esplicito che permetta di convertire in modo automatico uno Schema in un altro. Per risolvere questo ed altri problemi vedremo il linguaggio DAML+OIL (Defense Advanced Research Projects Agency Agent Mark-up Language + Ontology Inference Layer) che altro non è che una ontologia costruita su RDF Schema che fornisce meccanismi base di integrazione tra ontologie.

[2] Il naturalista svedese Carl von Linné (Räshult 1707 - Uppsala 1778) introdusse il metodo sistematico o classificatorio nello studio delle forme viventi suddividendole gerarchicamente in classi, ordini, generi e specie.

1.2.9 Lo strato logico

Come più volte detto, fino al livello delle ontologie e quindi anche al livello di DAML+OIL non vi è inferenza, solo rappresentazione della conoscenza.

A questo punto, avendo documenti scritti in RDF che fanno asserzioni e citazioni su risorse utilizzando proprietà definite in schemi tra loro resi compatibili tramite altre asserzioni che usano proprietà DAML+OIL, ci si interroga sul come utilizzare tutto ciò al fine di poter estrarre informazione utile da questi documenti.

Pensiamo al fatto che documenti di questo tipo saranno collegati tra loro sul Web e che quindi una ricerca all'interno di un documento o in generale sul Web prevede di collegare logicamente informazioni presenti su più documenti. La conoscenza di fatti quali `Pia è una mamma` e `Una mamma è un genitore` dovranno permettere di concludere logicamente che `Pia è un genitore`.

La generalità di questa struttura permette la realizzazione di ontologie che definiscano a loro volta regole che verranno *processate* dallo strato logico.

1.2.10 Lo strato della Fiducia: Web of trust

Abbiamo detto che RDF è il linguaggio base con il quale descrivere le nostre conoscenze sul mondo. Le pagine Web del domani saranno documenti RDF che tradurranno le conoscenza di dominio delle varie entità che pubblicheranno sul Web. Database già esistenti saranno tradotti per il Web semantico ed i loro contenuti saranno etichettati con delle URI in modo che altri possano fare delle asserzioni su di loro.

In base al principio che tramite RDF sul Web semantico *Qualsiasi cosa può dire qualsiasi cosa su qualsiasi cosa* si potranno creare situazioni *indesiderate*. Potrà infatti accadere che in una pagina vi sia l'asserzione *Franco ama il pesce* mentre in un'altra

che *Franco odia il pesce*. Essendo stato asserito che `ama` ed `odia` sono due proprietà tra loro incompatibili si verranno a creare due grossi problemi. Il primo è di ordine logico e lo affronteremo separatamente in seguito, il secondo è legato alla fiducia che l'utente dovrà avere nell'informazione che legge sul Web. Chi si potrebbe fidare di un sistema che permette di dire tutto ed il contrario di tutto?

Firma Digitale e *crittografia* permetteranno di dimostrare, appoggiandosi sullo strato logico, che una certa asserzione è stata fatta effettivamente da una certa entità. A questo punto sarà l'utente a dire di quali firme digitali *fidarsi*. Sarà possibile *settare*[3] il proprio livello di accettazione dell'informazione presente sul Web, filtrando tutto ciò di cui non ci fidiamo. Così procedendo, l'utente si fiderebbe di un numero ristretto di *enti* che conosce, numero che è infinitesimo rispetto a quelli che possono essere considerati affidabili, nell'intero Web. Da qui la questione di come *allargare* il numero di enti da considerarsi attendibili per un dato utente.

L'idea è quindi quella di creare il *Web of Trust* che è uno degli obiettivi primari del Web semantico. Ad esempio se Franco si fida di Alessandro ed Alessandro si fida di Angela si potrà fissare che la fiducia è una proprietà transitiva, ottenendo che Franco si fida di Angela, legando quindi gli utenti fra loro in una rete di fiducia. Questa rete di fiducia creerà dei grafi in cui le persone al loro interno avranno fiducia tra loro. Il livello di fiducia potrebbe essere pesato rispetto a qualche parametro, ad esempio una distanza topologica nel grafo della fiducia.

Così come fissiamo relazioni di fiducia così possiamo fissare relazioni di sfiducia e considerare anche questa proprietà facente parte della rete di fiducia a cui noi apparteniamo. Se durante una esplorazione si dovesse incontrare un documento del quale non

[3]Etim.: dall'inglese to set.

abbiamo prova esplicita né di fiducia né di sfiducia il nostro sistema potrebbe per default assumere di avere più fiducia in questo documento di quanta ne abbia per un documento per il quale esiste prova esplicita di sfiducia. In considerazione del fatto che l'utente, durante la navigazione, dovrebbe essere concentrato solo sul contenuto informativo della pagina, tutta la parte di *filtraggio* dell'informazione dovrebbe avvenire in modo automatico e trasparente.

Nell'ambito della comunità del Web semantico si parla del bottone `Oh yeah?`. Quando l'utente perde fiducia in quello che sta leggendo preme il bottone `Oh yeah?` facendo partire un motore inferenziale che a ritroso, partendo dai metadati, dia una dimostrazione di fiducia. Il risultato del bottone `Oh yeah?` potrebbe essere o una lista di assunzioni sulle quali si basa la fiducia o un messaggio di errore causato da una firma digitale non corretta o dalla mancanza di un percorso di fiducia tra l'utente e la pagina.

1.3 La rappresentazione della conoscenza

Uno degli storici sottocampi dell'intelligenza artificiale è sempre stato quello della KR (Knowlwdge Representation). Il campo della KR è sempre rimasto importante, tanto da generare a sua volta altri sottocampi di ricerca. Un campo però che è sempre rimasto attivo nella ricerca è quello della realizzazione di un linguaggio la cui sintassi fosse leggibile dalla macchina ma con una potente semantica formale definita dal linguaggio stesso. Così come le applicazioni per le quali la KR è utilizzata sono cresciute di complessità, così i linguaggi di rappresentazione della conoscenza sono cresciuti in capacità espressive. I primi linguaggi, quali KL-ONE e KRL, si sono evoluti verso linguaggi via via sempre più sofisticati quali LOOM, Classic e CYC-L.

Una delle linee guida nello sviluppo di nuovi linguaggi è sempre stata l'idea che il miglioramento di un linguaggio di KR dovesse andare nella direzione del miglioramento della sua capacità espressiva. Ognuno di questi linguaggi vuole rappresentare infatti informazioni concettuali generali ed è utilizzato per la costruzione della base di conoscenza di una singola entità ragionante. La KB (Knowledge Base) ottenuta può essere pensata come l'insieme delle credenze di questa singola entità. Un linguaggio di KR fornisce quindi un mezzo per esprimere ed esplicitare insiemi di credenze di un singolo agente razionale. Questa visione del mondo, in cui una KB rappresenta l'intero insieme di credenze che un entità ragionante ha su di un determinato dominio, porta alla necessità di linguaggi con la capacità di fornire strumenti per esprimere profondi legami relazionali tra oggetti e che forniscano allo stesso tempo mezzi per realizzare dell'inferenza sulla KB stessa.

Anche se tutto ciò è vero, si sta formando l'idea che per quanto riguarda Internet ed in particolare il World Wide Web, questo punto di vista debba essere cambiato. Ci si chiede quindi come dovrebbe essere un linguaggio di KR rivisto nell'ottica del WWW. Da un certo punto di vista il WWW è di per sé una KB: una massa enorme di informazioni che possono essere raccolte da un agente e sulle quali si possono effettuare interrogazioni.

1.3.1 Il Web visto come una base conoscenza

Da alcune stime, il numero di pagine sul web ha superato di gran lunga il mezzo miliardo ed il processo di crescita sembra inarrestabile. Se pensiamo che ogni pagina contenga anche solo una piccola informazione utile per essere considerata ci si rende facilmente conto che la mole di dati raccolti in questa unica KB metterebbe in crisi

qualsiasi sistema di KR. Questo è dovuto al fatto che molti dei linguaggi attuali di KR hanno aumentato la loro capacità espressiva a scapito della complessità computazionale. È infatti noto che gli attuali motori inferenziali che stanno alla base di molti linguaggi di KR hanno complessità computazionale NP-hard. Si deduce che un sistema di KR, per poter essere scalato alle dimensioni del Web, deve essere di gran lunga più efficiente di quanto non lo siano gli attuali. Avremo quindi che un nuovo sistema di KR dovrebbe fare delle scelte in modo da trovare un nuovo trade-off tra espressività ed efficienza. Un altro grosso problema è quello che difficilmente è pensabile un agente che sia in grado di raccogliere tutte le informazioni presenti sul Web. Lo stesso motore di ricerca di AltaVista indicizza molto meno della metà delle pagine presenti sul Web. Questo ci conduce al fatto che dovremmo sempre lavorare su di una KB incompleta (visto che non abbiamo raccolto tutte le informazioni presenti vi può essere qualche cosa di vero che è detto all'interno di qualche pagina che non abbiamo visto). Ricordiamo però che molti sistemi di KR si fondano sull'assunzione di mondo chiuso, ossia che tutto ciò che non è detto o non è deducibile dalla KB è considerato falso.

Vi sono delle soluzioni proposte che, in contrapposizione alla poco praticabile assunzione di mondo chiuso (Closed World Assumption), propongono sistemi di KR che si basano su un'ipotesi meno forte che è quella di LCW (Localized Closed World). L'idea, al fine di migliorare l'efficienza, è quella di considerare di aver una sorgente di informazioni che sia totale rispetto ad un argomento dato. Sicuramente la LCW è più ragionevole della CWA (Closed World Assumption) per il Web, ma resta ugualmente il problema di raccogliere tutte le informazioni su di un dato argomento avendo la garanzia che siano proprio tutte. La dinamicità del Web dà poi il colpo di grazia.

Certe pagine sono statiche nel tempo, altre cambiano ma senza modificare il loro contenuto semantico, altre nascono nuove mentre altre vengono rimosse. In una situazione del genere l'unica assunzione che un sistema di KR può fare è che la sua KB sia sempre *out-of-date*.

Un problema qui strettamente correlato è quello delle ontologie che, se non unificate all'interno di una precisa e condivisa cornice concettuale, rischierebbero di creare *linguaggi* incomprensibili ad agenti che volessero evincere informazioni dal Web. Il problema però di una alta standardizzazione delle ontologie è quello di creare un recinto così rigido tale da non permettere a nuove idee, nuove conoscenze o nuovi argomenti di essere facilmente introdotti sul Web in un modo da renderli fruibili immediatamente da tutti gli agenti. Si dovrà quindi creare un sistema di ontologie che sia sufficientemente flessibile per renderlo incrementale nel tempo.

La mancanza di un controllo centralizzato che governi Internet, ossia la mancanza di vincoli di integrità, fa sì che non si possa mai considerare una pagina affidabile e nemmeno persistente. Tutto ciò conduce inevitabilmente ad inconsistenza delle informazioni. Certe inconsistenze saranno da considerarsi degli errori altre potranno essere imputate a differenti punti di vista degli autori.

1.3.2 Le reti semantiche

Rivediamo alcuni concetti sulla rappresentazione della conoscenza che ci condurranno spontaneamente a capire perché siamo interessati ad RDF come linguaggio per descrivere ciò che conosciamo sul mondo, ed accedervi.

Ricordiamo il percorso fatto sulla rappresentazione della conoscenza. L'idea è quella che esiste un mondo, quello reale. Di questo siamo interessati ad alcuni oggetti

materiali o immateriali che vi appartengono, nel senso che siamo interessati a trattare per mezzo di un sistema automatico di deduzione una parte per noi interessante della realtà sulla quale fare delle deduzioni utili. Quello che dobbiamo fare è un atto ontologico attraverso il quale decidere quali sono gli oggetti a cui siamo interessati e quali siano le proprietà di cui questi oggetti godono.

Vediamo un primo semplice esempio. Pensiamo di essere interessati a descrivere una macchina in termini del suo colore rosso. Avremo quindi che l'oggetto `macchina` gode di una certa proprietà, la *rossezza*. Possiamo rappresentare questa situazione attraverso la proprietà `red` e l'oggetto `macchina` asserendo in Prolog che `red(macchina)`.

Ma questo tipo di rappresentazione della nostra conoscenza sul mondo reale non è abbastanza elastica per le nostre esigenze, nel senso che non saremo in grado di fare domande del tipo *Di che colore è la* ***macchina****?* a meno di non introdurre meccanismi per logiche dell'ordine superiore, con le note implicazioni di complessità concettuale e computazionale.

Per ovviare a ciò possiamo pensare di *trasformare* la proprietà `red` in un nome di individuo e di introdurre un nuovo predicato `colore`. Avremo quindi che `colore(red,macchina)` esprime il fatto che l'individuo `macchina` è legato all'individuo `rosso` attraverso la relazione `colore`. Rifrasato *La macchina è rossa.* Ma in questo modo non siamo in grado di rispondere a domande del tipo *Quale proprietà della macchina* ***macchina*** *ha valore* ***rosso****?*.

Questo ci porta alla formalizzazione più generale nella quale si introduce una generica proprietà `prop(Obj,Att,Val)` in cui si dice che l'oggetto `Obj` ha valore `Val` per l'attributo `Att`.

Definizione 1.3.1. La *reificazione* [10] è l'atto della trasformazione di un nome di proprietà in un nome di individuo.

Possiamo pensare di descrivere la conoscenza che abbiamo sulla realtà attraverso triple del tipo `(Obj,Att,Val)` in cui individuiamo nel dominio materiale degli oggetti `Obj` che godono di proprietà `Att` che assumono un certo valore `Val`? Se ad esempio volessimo dire che *Franco è uno studente* come faremmo in termini di una tripla? L'idea è quella di introdurre un predicato speciale `is_a` che ha come semantica quello di dire che un dato oggetto è una certa cosa (dove omettiamo di addentrarci nel significato di cosa significa essere qualche cosa). Volendo possiamo pensare di rifrasare l'affermazione qui sopra dicendo che *Franco è istanza alla classe degli studenti*, riportando la semantica dell'essere a quella dell'appartenenza ad insieme. Nell'approccio `(Obj,Att,Val)` avremo che `prop(franco,is_a,stundent)` rappresenta nella sintassi Prolog la conoscenza che abbiamo sul nostro dominio materiale.

In situazioni nelle quali vi sia una certa proprietà con molti valori, ci si può sempre ridurre alla situazione `(Obj,Att,Val)`. Fra l'altro questa rappresentazione ha il grande vantaggio di essere incrementale, nel senso che nuova conoscenza può essere facilmente aggiunta alla nostra KB senza modificare le terne costruite precedentemente.

In questo discorso abbiamo sottaciuto il fatto che, visto che l'unica proprietà usata è la proprietà fittizia `prop`, essa può essere omessa senza perdita di generalità. Ciò conduce immediatamente ad una rappresentazione grafica della nostra KB detta *rete semantica*. Si hanno cioè dei grafi orientati in cui gli archi sono etichettati con attributi, il nodo di partenza è l'oggetto ed il nodo di arrivo è il valore. La cosa interessante di questo modo di rappresentare la conoscenza è che il valore a sua volta può essere visto

come l'oggetto di una nuova tripla. Questa tecnica permette quindi di rappresentare situazioni anche molto articolate. Ad esempio, si può dire `(franco,scrive,tesi)` e `(tesi,scritta_in,latex)` in cui `tesi` è valore e poi oggetto da cui *Franco scrive la tesi in Latex*.

Il predicato speciale `is_a` permette di realizzare ereditarietà ossia la possibilità di derivare nuova conoscenza dai fatti noti.

Esempio 1.3.1. Pensiamo di conoscere i seguenti fatti:

```
prop(obj,is_a,Class_1).
prop(Class_1,att_1,val_1).
prop(Class,att_2,val_2).
prop(Class_1,is_a,SuperClass_1).
prop(SuperClass_1,att_3,val_3).
```

osservando il grafo in figura 1.1 a pagina 29 possiamo dedurre nuova conoscenza e dire che:

```
prop(obj,att_1,val_1).
prop(obj,att_2,val_2).
prop(obj,att_3,val_3).
```

il tutto in virtù della semantica del predicato speciale `is_a`.

Ritorniamo al fatto che pensiamo di descrivere la nostra conoscenza sul mondo attraverso triple e ricordiamo che il primo passo prima di pensare di fare deduzione automatica è quello di rappresentare la conoscenza. Vedremo che in ultima analisi RDF è un linguaggio per descrivere il *mondo* in termini di triple e quindi può essere considerato un linguaggio per la costruzione di reti semantiche sul Web. Potremo quindi considerare il Web semantico come una *gigantesca* rete semantica distribuita sulla quale nessuno ha un controllo globale, ma solo locale, totalmente descritta attraverso delle asserzioni RDF.

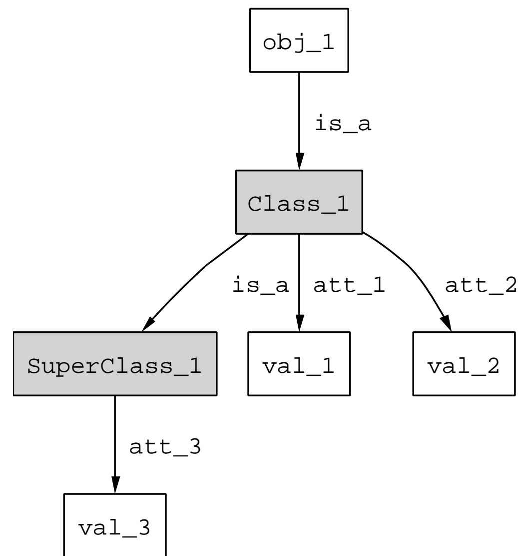


Figura 1.1: Un esempio di rete semantica [esempio: 1.3.1].

1.4 Il problema dei nomi

Abbiamo detto che sul Web semantico non è possibile fare l'UNA (Unique Name Assumption) e che è stato scelto il meccanismo delle URI per assegnare nomi. Vediamo ora più in dettaglio questo meccanismo.

1.4.1 Le URI

Se sul Web semantico vogliamo descrivere il mondo in termini di relazioni tra oggetti, prima di tutto dovremo capire come oggetti e relazioni possono essere nominati in modo non ambiguo. Avremo infatti che il nome `Franco` non può essere considerato un buon nome per riferirsi a me visto che, in un contesto di scala mondiale come il Web, di persone alle quali ci si potrebbe riferire con il nome `Franco` ce ne potrebbero essere delle altre, creando quindi ambiguità.

Gli oggetti di cui potremmo voler parlare possono essere persone, oggetti, idee o persino affermazioni. Mentre le relazioni tra questi possono essere appartenenza, colore o qualsiasi altra cosa.

Per eliminare l'ambiguità dei nomi sul Web Semantico si utilizzano le URI (Universal Resource Identifiers). Noi possiamo assegnare una URI a qualsiasi cosa e qualsiasi cosa che abbia una URI si può dire che *è sul Web*. Tutto può avere una URI, anche *il ronzio del processore del mio PentiumIII* può essere associato ad una URI.

Una delle forme più note di URI è la URL (Universal Resource Locator) che specifica l'indirizzo di una pagina sul Web (e.g. `http://www.unimi.dsi.it/`). A differenza di molte altre forme di URI una URL allo stesso tempo identifica e localizza una risorsa. Avremo infatti che attraverso la sola identificazione di una risorsa, non saremo però in grado di localizzarla.

Il controllo delle URI è quasi completamente decentralizzato, ogni entità può generarne di proprie. Lo schema URI `http:` che definisce le URL, è invece controllato in modo centralizzato dal DNS (Domain Name Server), ciò garantisce che non vi sia duplicazione di nomi. Mentre altri schemi di URI quali quello dei `freenet:` sono completamente decentralizzati creando quindi il problema che potrebbe esistere lo stesso nome per due oggetti diversi. Per tutte le URI c'è però il problema che è possibile avere URI diverse che si riferiscono allo stesso oggetto. Ma cosa ben peggiore, non sarà possibile capire se due URI si riferiscano alla stessa risorsa. Cade infatti l'assunzione di nome unico, in cui si presume che ogni oggetto del dominio del discorso sia nominato attraverso un solo *termine*.

Un semplice modo per creare una URI è quello di partire da una pagina Web. In questa pagina si *descrive* l'oggetto che si vuole identificare e si dice che la sua

URI (il suo nome per il Web semantico) è la URL della pagina stessa. Per esempio se voglio creare una URI che identifichi *il PC su cui in questo momento sto scrivendo la tesi* non faccio altro che costruire una pagina domiciliata alla URL `http://example.com/MyPentium` in cui, al suo interno, *dico* che *La URI uguale alla URL* `http://example.com/MyPentium` *identifica il PC localizzato nella sala macchine del DSI dell'Università degli Studi di Milano con matricola pari a 2586.*

È interessante notare che la URI `http://example.com/MyPentium` svolge contemporaneamente due diversi scopi. Il primo è quello di identificare (anche se in modo non univoco) il computer fisico sul quale sto scrivendo la tesi, il secondo identificare univocamente la pagina Web nella quale è descritto.

Quello che è importante capire è che una URI non è un indirizzo che permette ad un browser di trovare un file sul Web ma è il nome di una risorsa. Questa risorsa può o non può essere accessibile via Internet. La URI può o non può provvedere a informazioni aggiuntive sull'oggetto che identifica. È altresì vero che una URL è una URI che fornisce un modo per trovare maggiori informazioni relative ad una risorsa e talvolta ritrovare la risorsa stessa.

Il problema dell'identificazione delle risorse sul Web è un problema aperto che ricade sotto la tematica del *Semantic Web Identification Problem*. Deve essere però essere chiaro che una URI è solo un nome di una risorsa e null'altro, non si deve fare nessuna altra considerazione aggiuntiva sul fatto che un nome possa essere una URL e che quindi la risorsa sia la URL stessa, potrebbe infatti essere che la URL identifichi solo la descrizione della risorsa e non la risorsa stessa.

Come risulterà più chiaro dopo aver parlato di RDF la descrizione stessa dell'oggetto di cui la URL ne è il nome avverrà proprio attraverso asserzioni fatte usando RDF

XML. Sarà chiarito altresì che il nome di una risorsa sarà dato da una URL seguita dal simbolo # seguito a sua volta dal nome vero e proprio. Ogni nome diverso di questo tipo identificherà una risorsa diversa. Questo meccanismo permetterà di *domiciliare* nomi diversi alla stessa URL ossia nella stessa pagina.

1.5 Il mark-up

Vediamo cosa si intende per mark-up attraverso un semplice esempio. All'interno di un qualsiasi testo scritto gli *spazi*, i *punti*, le *virgole* ed i *margini* non fanno parte del testo, ma del mark-up. Servono per permettere al lettore di istituire dei comportamenti *paralinguistici* in loro funzione. Un punto ad esempio permetterà di cambiare l'intonazione della frase aumentandone la comprensibilità, mentre è noto che una virgola ne può modificare il significato. Ecco quindi che il mark-up dei testi serve per migliorarne la leggibilità e più in generale la comprensibilità. È facile intuire che essendo il mark-up fortemente legato alla semantica di un testo, sarà proprio attraverso un sofisticato meccanismo di *marcatura* che verrà fornita una semantica comprensibile ad agenti software.

Definizione 1.5.1. Per *tag* si intende ciò che all'interno di un testo non fa parte del linguaggio che il mark-up descrive. È inteso definire particolari significati paralinguistici di parte del testo che esso marca.

Definizione 1.5.2. Per *browser* si intende un qualsiasi oggetto software che, avuto in ingresso un testo marcato, lo visualizza rispettando però la semantica intesa del mark-up in esso contenuto.

1.5.1 Tipi di mark-up

Esistono modi diversi per marcare un testo, vediamone i più importanti.

Puntuazionale

Attraverso un insieme di segni tipografici standard si veicolano informazioni di carattere sintattico. L'interpretazione dei simboli è stabile nel tempo e ben nota, anche se sono risaputi problemi relativi al suo uso: l'utilizzo delle virgolette aperte e chiuse o di quelle neutre, il punto (.) indica sia un fine periodo che una abbreviazione.

Presentazionale

Consiste in un insieme di *effetti* che servono per migliorare la presentazione. Ad esempio un inizio di paragrafo su nuova linea, un cambio pagina condizionato all'inizio di ogni nuovo capitolo o i caratteri per indicare un elenco per punti.

Procedurale

Sono dei comandi o delle procedure che devono essere attivate al fine di ottenere una certa visualizzazione (e.g. i comandi *punto* di WordStar). I file RTF (Rich Text Format) contengono al loro interno marcature di tipo procedurale, nel senso che è specificato cosa deve essere fatto per ottenere la visualizzazione voluta. Si può pensare di voler dire *questo* `testo` *deve essere visualizzato in Courier 12 punti.*

Descrittivo

Sono marcatori che indicano il ruolo di quella parte di testo (e.g. questo è un titolo o questa è una poesia). Può essere fatto per categorie, ad esempio ci possono essere più

tipi di titoli. Ovviamente non viene specificato il come viene visualizzato ma solo il ruolo di quella parte di testo all'interno del testo stesso, demandando poi al *browser* la vera e propria visualizzazione solo nel momento della visualizzazione stessa.

Referenziale

Quando si utilizzano dei marcatori che fanno riferimento a qualche cosa di *esterno* al testo stesso (e.g. l'utilizzo di una sigla che verrà poi espansa automaticamente).

Dichiarativo

Definisce le regole e la struttura che forniscono gli strumenti linguistici per l'introduzione e l'interpretazione di nuovi simboli di mark-up. Grazie a queste regole sarà possibile controllare se un testo sia stato marcato correttamente rispetto alla struttura.

1.5.2 Visualizzazione del mark-up

Al di là degli scopi che si prefigge il mark-up, questo può essere visualizzato all'interno del testo in modi diversi a seconda dell'uso e della situazione. Può infatti essere visualizzato il mark-up, ma allo stesso tempo il suo effetto. A seconda delle situazioni può essere parte integrante del testo stesso in modo quasi indistinguibile (punteggiatura), o mantenuto nel testo stesso ma delimitato da caratteri speciali per distinguere quale parte è testo e quale parte è mark-up o ancora, alternativamente, può essere conservato in un file separato distinguendo quindi a monte la parte di testo da quella di mark-up. I possibili comportamenti di un sistema nei confronti di un testo con mark-up possono essere i seguenti:

Esposto

Mark-up ed effetto del mark-up sono visualizzati.

Travestito

Viene visualizzato il mark-up attraverso dei simboli.

Nascosto

Non viene visualizzato il mark-up ma solo i suoi effetti.

Visualizzato

Testo e mark-up vengono visualizzati contemporaneamente.

1.5.3 Le componenti del mark-up

All'interno di un testo marcato possiamo individuare vari *oggetti* che svolgono ruoli diversi.

Elementi

Sono le parti di testo che hanno un senso proprio ossia che ci possono considerare, a livello di interpretazione, come atomici. Ad esempio un titolo è un *elemento* e come tale può essere marcato, attraverso opportuni tag, dicendo che quella stringa è un titolo.

Attributi

Sono informazioni relative all'elemento che però non sono parte dell'elemento. Sono contenute normalmente nel tag di marcatura. Ad esempio in:

```
<capitolo N=1>Introduzione</capitolo>
```

avremo che: `capitolo` è il tag, `Introduzione` è l'elemento, `N` è l'attributo ed `1` è il valore dell'attributo.

Entità

Sono parti di documento memorizzate separatamente e richiamabili all'interno del documento stesso. Possiamo pensare a loro come a dei nomi che verranno espansi in modo automatico. Ad esempio si potrebbe definire che [Ws] è una entità che sarà di volta in volta espansa in *Web semantico*.

Parsed Character Data

Sono i caratteri veri e propri del testo.

Commenti

Sono una parte di testo che non viene né interpretata né visualizzata.

Processing Instruction

Sono comandi espliciti come ad esempio un fine pagina.

1.6 I linguaggi di mark-up

La problematica di *marcare* un testo era sentita già in una fase pre-Web, ed è per questo che nasce tutta una serie di linguaggi e metodologie per marcare testi al fine di permettere l'interoperabilità.

Da sempre le grandi aziende hanno inseguito un formato aperto e standardizzato per il trattamento dei dati che desse la possibilità di scambiare e manipolare documenti. Ora questa strada sembra essere stata trovata con l'XML (eXtensible Mark-up Language), un metalinguaggio in grado di sintetizzare in una nuova struttura i vantaggi che potevano offrire da una parte l'HTML, che ormai mostra i suoi limiti, dall'altra il più vecchio SGML (Standard Generalized Makup Language).

Ma per arrivare all'XML, è opportuno ripercorrere brevemente la storia dell'ipertestualità. È negli anni Settanta che si cominciano ad avere i primi risultati verso la standardizzazione del trattamento dei dati: prima con il linguaggio GenCode creato dalla Graphic communications association, poi con l'IBM (International Business Machine), che sviluppa il Generalized Mark-up Language (GML - Generalized Makup Language, basato su una sintassi piuttosto semplice di tag, ovvero di marcatori contenuti tra <> in apertura e </> in chiusura) nel tentativo di risolvere i propri problemi interni per la pubblicazione di manuali ed altre informazioni.

Agli inizi degli anni Ottanta gli sviluppatori GenCode si uniscono a quelli GML per formare un comitato nell'American National Standards Institute (ANSI), dal quale nel 1986 nasce lo Standardized Generalized Mark-up Language (SGML). Tra coloro che cominciarono ad utilizzare questo nuovo mark-up c'era anche Tim Berners-Lee, che alla fine degli anni Ottanta prese da un DTD (Data Type Definition) SGML, un assortimento di tag, ai quali aggiunse un particolare meccanismo di link. Con questa

intuizione nasce l'ormai famosissimo HTML, che diventa il linguaggio di mark-up per le applicazioni Web nel momento in cui vengono introdotti i browser.

1.6.1 HTML

In effetti lo scenario di Internet cambia molto velocemente e così standard correlati che erano stati concepiti per altri scopi, come appunto l'HTML, adesso mostrano la corda. Perfino il protocollo IP (Internet Protocol), forse uno degli standard più longevi della storia dell'informatica, è in corso di revisione e questo è forse l'immagine più evidente del salto che Internet ha compiuto nel corso degli ultimi anni, ma il cambiamento che sta avvenendo a livello applicativo è molto più radicale: la staticità degli standard fin qui definiti da fattori di successo tende a diventare sempre più un peso, che solo una maggiore duttilità dei nuovi standard può compensare.

Tutto ciò è dovuto ad alcuni limiti insiti nell'HTML, il linguaggio che ha fatto dilagare il Web ma che per esempio non potrà mai imporsi come linguaggio universale per la costruzione di applicazioni sul Web semantico. L'HTML è infatti un linguaggio nato sostanzialmente per un *publishing* elementare, ma ben presto ha dovuto lasciare spazio allo sviluppo di tecnologie parallele che potessero assicurare in fondo la sua sopravvivenza: stiamo parlando dei vari Javascript e plugin tipo Shockvawe o Acrobat reader, che in pratica hanno trasformato l'HTML in un assemblatore di tecnologie, piuttosto che un linguaggio vero e proprio.

Questo ha comportato anche problemi di portabilità delle applicazioni Web, perché tutte queste nuove tecnologie non sono standard, bensì soluzioni private più o meno diffuse, che necessitano per essere utilizzate di software particolari o di particolari versioni di esso. Tutto ciò è lontanissimo dalla filosofia iniziale ma anche dalla tendenza

attuale che punta a costruire ambienti standard in grado di permettere la costruzione di applicazioni portabili a prescindere dal sistema operativo o dal tool di sviluppo.

Ma i problemi di portabilità non sono gli unici ad affliggere il mondo dell'HTML, che deve scontare anche la sua enorme rigidità, nonostante il consorzio W3C rilasci periodicamente delle specifiche sulle quali vengono progettati i browser e i tool di sviluppo. Con HTML lo sviluppatore è vincolato all'uso dei tag definiti nelle specifiche rilasciate dal consorzio, nel caso di SGML è invece possibile definire i tag a proprio piacimento.

1.6.2 XML

Ecco che allora a metà strada fra l'HTML, (semplice ma troppo legato alla visualizzazione e al livello grafico, senza la possibilità di trasmettere dati) e l'SGML, appare finalmente il linguaggio XML, che il W3C ha definito in questo come:

> "*The eXtensible Mark-up Language (XML) is a subset of SGML. It's goal is to enable generic SGML to be served, received and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML*".

Rispetto ad HTML, il nuovo metalinguaggio può vantare tre importanti vantaggi:

1. l'estensibilità, che permette la definizione di set personalizzati di tag;

2. la salvaguardia degli elementi strutturali definiti in un file esterno chiamato Document Type Definition (DTD);

3. la validazione per la quale ogni documento passa attraverso un controllo che ne attesta la conformità alle regole definite nel DTD.

Il W3C ha quindi optato per la definizione di un linguaggio che, pur mantenendo tutti i vantaggi dell'SGML, fosse più semplice e quindi più adatto alla varietà di sviluppatori di documentazione su Internet. Nel febbraio del 1998 il W3C ha rilasciato la versione 1.0 delle specifiche di XML, che fra l'altro prevede la definizione di un gruppo di specifiche, che raccoglie anche l'XLL per la gestione dei link e l'XSL (Extensible Stylesheet Language) per la rappresentazione.

Ecco perché si può quindi parlare di metalinguaggio, che serve a definire altri linguaggi o applicazioni. Un tool per leggere un documento XML consta di due parti:

1. il *Parser* esegue il controllo semantico e gestisce gli errori;

2. il *Processor* utilizza un altro file in cui è definita la formattazione dei vari tag, per visualizzare il documento.

Già da questa dinamica si capisce come la separazione fra struttura e rappresentazione, che come si è visto è uno degli aspetti chiave della buona costruzione di un ipertesto, sia garantita attraverso la separazione fisica dei dati che governano i due aspetti ed addirittura attraverso la separazione dei linguaggi (la rappresentazione viene infatti gestita dal XSL).

Una breve introduzione alla sintassi XML

XML come abbiamo già accennato è una versione semplificata di SGML e risulta quindi essere un metalinguaggio di mark-up. Siamo qui interessati al fatto che XML servirà come linguaggio in cui esprimere i modelli RDF di cui si parlerà più avanti

in questa tesi. Risulta quindi fondamentale capire almeno a livello elementare la *grammatica* di XML e con essa la struttura di un documento scritto in XML.

XML permette di definire nuovi linguaggi di mark-up attraverso Elementi, Attributi, Entità, #PCDATA, Commenti, Processing Instruction. Un documento XML generalmente si *appoggia* su di un DTD (Data Type Definition) all'interno del quale sono contenute informazioni sulla struttura dei tag che serviranno per marcare il testo contenuto nel documento XML.

Un documento XML è strutturato in tre parti:

1. dichiarazione XML;
2. dichiarazione di tipo;
3. istanza.

La *XML declaration* è contenuta all'inizio di un documento e contiene fra le altre informazioni sulla versione. È il prologo del documento in cui è contenuto un tag speciale che dice che quello è un documento XML `<?xml version 1.0?>`

La *type declaration* contiene le regole che permettono di effettuare una verifica strutturale del documento. Ciò avviene attraverso `<!DOCTYPE...>`, che permette di specificare dove sono contenute le informazioni del DTD. È una sorta di modellizzazione della classe, possiamo infatti pensare alla strutturazione dei tag come ad un oggetto che viene via via specificato nei termini delle sue componenti base. Il DTD può essere su di un file esterno o in parte contenuto nel documento stesso.

L'*Istanza* contiene il documento vero e proprio, marcato con i tag la cui struttura è specificata nel DTD. In questo modo si potrà effettuare una verifica sull'istanza stessa dicendo se ci sono state violazioni rispetto al suo DTD. La struttura di questa parte

del documento prevede sempre l'esistenza di un elemento *radice* che esiste sempre ed è unico per quel documento.
Relativamente alla validazione un documento XML si dice:

- *valido* se rispetta le regole specificate nel DTD;
- *ben formattato* se si presenta con una struttura sufficientemente regolare da essere controllata. Devono cioè essere rispettate regole di carattere molto generale (e.g. ogni tag di apertura deve essere associato un tag di chiusura).

Un file DTD conterrà al suo interno la definizione del linguaggio di mark-up, data attraverso:

- elementi definiti attraverso il tag speciale <!ELEMENT...>
- attributi definiti attraverso il tag speciale <!ATTLIST...>
- entità definite attraverso il tag speciale <!ENTITY...>

Quando si definisce un *elemento* tramite <!ELEMENT...> si definisce il suo *tag*, ciò avviene attraverso:

```
<!ELEMENT nome content-model>
```

Dove nome è il nome del nuovo tag che stiamo definendo (e.g. autore), mentre content-model serve per *descrivere* quale deve essere il contenuto marcato dal tag nome. Vediamo i *valori* che può assumere content-model e di riflesso i valori che può assumere l'elemento marcato:

- ANY può contenere qualsiasi elemento definito nel DTD oppure del puro testo;

- EMPTY non ha contenuto;
- #PCDATA puro testo;
- (...) può contenere solo altri elementi specificati in parentesi, secondo una sintassi che vedremo dopo;
- vi è poi un caso *misto* in cui può contenere sia testo che altri elementi (l'unico modo per utilizzare questa opportunità è quello di costruire una lista di alternative con un asterisco alla fine ed in cui il primo elemento è un #PCDATA).

Abbiamo visto che un elemento può essere definito in modo da poter contenere altri elementi che a loro volta saranno stati definiti altrove. Il nome degli elementi che devono o possono essere contenuti è specificato seguendo la seguente sintassi:

- la virgola (`,`) indica che devono essere presenti entrambi gli elementi;
- la pipe (`|`) indica un or esclusivo di presenza di due elementi.

Al temine delle parentesi tonde possono essere presenti caratteri diversi con diversi significati:

- *nessun carattere* indica che la lista deve comparire una ed una sola volta;
- `?` indica che la lista può apparire o non apparire;
- `+` indica che la lista deve apparire almeno una volta;
- `*` indica che la lista deve apparire zero o più volte.

Relativamente ad un tag si possono specificare degli attributi che sono informazioni sugli elementi che però non fanno parte del testo stesso. Gli attributi possono essere pensati talvolta come dei parametri, ad esempio un tag di allineamento potrebbe avere un attributo che può assumere i valori `sinistra`, `destra` o `centrato` per poter definire come il testo dovrà essere allineato. Nel file DTD attraverso `<!ATTLIST...>` si possono specificare quali sono i valori leciti degli attributi di un dato tag. In generale la sintassi è la seguente:

```
<!ATTLIST nome nomeAttributo tipo default,...>
```

In cui `nome` è un nome di tag definito tramite il tag speciale `<!ELEMENT...>` mentre `nomeAttributo` è il nome dell'attributo che stiamo definendo relativo al tag `nome`. Vediamo ora attraverso degli esempi l'utilizzo di `tipo` e `default` che servono per specificare quali valori può o deve assumere un dato attributo.

```
<!ATTLIST nome att CDATA "due">
```

In questo caso l'attributo di nome `att` relativo all'elemento di nome `nome`, avendo come tipo `CDATA`, può assumere qualsiasi valore. Se non fosse specificato nessun valore per l'attributo `att` questi assumerebbe per default il valore `due`.

```
<!ATTLIST nome att (uno|due|tre) "tre">
```

In questo caso l'attributo di nome `att` relativo all'elemento di nome `nome`, avendo come tipo `(uno|due|tre)`, può assumere qualsiasi valore compreso nella lista data. Se non fosse specificato nessun valore per l'attributo `att` questi assumerebbe per default il valore `tre`.

```
<!ATTLIST nome att ID #IMPLIED>
```

In questo caso l'attributo di nome `att` relativo all'elemento di nome `nome`, avendo come tipo `ID` e come default `#IMPLIED`, può assumere un unico valore in tutto il documento.

```
<!ATTLIST nome att IDREF #IMPLIED>
```

In questo caso l'attributo di nome `att` relativo all'elemento di nome `nome`, avendo come tipo `IDREF` e come default `#IMPLIED`, può assumere un valore assunto da un attributo di tipo `ID` in un qualche punto del documento.

In generale un attributo può o non può comparire nel tag di apertura di un elemento.

I valori di default degli attributi sono:

- `valore`: è il valore che deve assumere quell'attributo se non diversamente specificato;
- `#REQUIRED`: quell'attributo è obbligatorio;
- `#IMPLIED`: facoltativo;
- `#FIXED`: esplicito e non modificabile.

Alcuni nomi di attributi sono *predefiniti* e normalmente iniziano con il *prefisso* `xml:` e sono:

- `xml:lang=language` per dire che il testo è scritto nel linguaggio `language`;
- `xml:space` per la gestione degli spazi;

- `xml:link`;
- `xml:attribute`;

Abbiamo già parlato delle *entità* come oggetti del mark-up, una entità può essere definita attraverso la sintassi:

```
<!ENTITY nome "valore">
```

Dove `nome` rappresenta il nome dell'entità che verrà usata nel file XML mentre `valore` rappresenta la stringa di caratteri che verrà sostituita. Se `valore` assume valore `#n`, dove `n` è un numero decimale, allora `valore` rappresenta il carattere ASCII della tabella dei caratteri attiva di valore `n`.

Per potersi riferire ad una *entità* all'interno di un documento XML che abbia dichiarato di riferirsi ad un dato DTD che contenga `<!ENTITY nome valore»` si deve usare la sintassi `&nome;`.

Se si volesse definire una *entità* da utilizzare all'interno del DTD stesso basterebbe far precedere `nome` dal carattere speciale %. Per potervisi riferire all'interno del DTD basterà usare la sintassi `%nome`.

Come ultima cosa vediamo il significato di:

```
<![CDATA[ stringa ]]>
```

Serve per inserire stringhe contenenti caratteri speciali che non si vuole vengano interpretati all'interno di un file XML.

Dopo questa spiegazione vediamo un esempio di file XML con relativo DTD che usa ciò che abbiamo appena spiegato, con l'*animo* di riuscire a leggere il contenuto

di un file XML (cosa che risulterà utile per leggere file RDF scritti con la sintassi XML appunto). Prima di tutto vediamo il file `admission.dtd` che definisce i tag che andremo poi ad utilizzare nel file XML:

```
<!ENTITY %STD "#PCDATA">
<!ENTITY CS "Computer Science">
<!ELEMENT admissionStanford(req)*>
<!ELEMENT req(dept,name,year,(minscore|(minQ,minA, minV))>
<!ELEMENT dept %STD;>
<!ELEMENT name #PCDATA>
<!ELEMENT year #PCDATA>
<!ELEMENT minscore #PCDATA>
<!ATTLIST minscore type ("CBT" | "Paper Based") #REQUIRED>
<!ELEMENT minQ #PCDATA>
<!ELEMENT minA #PCDATA>
<!ELEMENT minV #PCDATA>
```

A questo punto avendo definito tag ed entità usiamoli pensando di marcare un file XML che parla dei *requirement* di ammissione a Stanford:

```
<?xml version="1.0"?>
<!DOCTYPE admissionStanford SYSTEM "admission.dtd">
<admissionStanford>
 <req>
  <dept>&CS;</dept>
  <name>TOEFL</name>
  <year>2002</year>
  <minscore type="CBT">230</minscore>
 </req>
 <req>
  <dept>&CS;</dept>
  <name>GRE</name>
  <year>2002</year>
  <minQ>770</minQ>
  <minA>720</minA>
```

```
  <minV>650</minV>
 </req>
</admissionStanford>
```

Questa non voleva essere una spiegazione esaustiva della sintassi e delle potenzialità di XML, ma solo una introduzione che fornisse gli strumenti minimi di comprensione per la sintassi RDF XML. Seguono a questo punto altre due importanti spiegazioni su XML che torneranno utili in seguito.

I namespace

Abbiamo visto che un tag è caratterizzato da un nome (e.g. `score`) e che un tag può avere degli attributi, ed ogni attributo un nome (e.g. `type`). Questa tecnica di dare i nomi può però portare a delle situazioni *sgradevoli*.

Esempio 1.6.1. Si vuole utilizzare un tag che ha lo stesso nome, ma un diverso significato, di un altro tag definito in un DTD diverso (e.g. quando si ha un tag `titolo` per marcare il titolo di un libro ed un altro tag `titolo` per marcare il *titolo* [Dott., Prof., ecc.] di una persona).

L'idea per ovviare a problemi di questo tipo è quella di creare nomi di tag e nomi di attributi preceduti da un prefisso che indichi il loro *namespace*. Quello che si otterrà saranno nomi del tipo:

```
prefisso:nome
```

In questo modo `prefisso` sarà il *nome* di uno spazio di nomi e `nome` sarà il *vero* nome del tag, anche se il nome del tag sarà a tutti gli effetti `prefisso:nome`.

Utilizzando questa strategia potremmo avere nomi del tipo `libro:titolo` (in cui `titolo` è un nome all'interno dello spazio dei nomi `libro`) e `peronaggi:titolo` in

cui `titolo` è un *nome* all'interno dello spazio dei nomi `personaggi`. Questo tipo di distinzione non serve solo per risolvere problemi di ambiguità ma anche per permettere al processore XML di trattare i due tag `titolo` in modo diverso.

Come è facilmente intuibile, un *namespace* dovrà essere a sua volta dichiarato prima di essere usato. La *dichiarazione* può avvenire in modi diversi, ma l'idea di fondo è sempre quella: collegare il `prefisso` ad una URI che sia stabile nel tempo. Essendo le URI degli oggetti univoci, non è possibile che ci siano sovrapposizioni tra i *prefissi*. Il limite sta però nel fatto che una URI può essere qualsiasi cosa, non deve essere per forza un link ad un DTD, è solo un'*ancora* ad un qualche cosa di univoco.

Un modo per dichiarare un nome di *namespace* è di farlo direttamente all'interno di un tag come se fosse un attributo (si usa il prefisso speciale `xmlns` che serve proprio per definire nomi di *namespace*):

```
<tag xmlns:prefisso="URI">
 [Blocco]
</tag>
```

A questo punto all'interno del `[Blocco]` si potrà *usare* il nome di namespace `prefisso` come prefisso per i nomi di tag ed attributi.

Un modo alternativo per dichiarare un nome di namespace è:

```
<tag xmlns="URI">
 [Blocco]
</tag>
```

All'interno del `[Blocco]` tutti i nomi sono riferiti per default allo spazio dei nomi identificato dalla URI, anche se non hanno nessun prefisso. Questo tecnica permette di evitare di dover mettere il *prefisso* a tutti i nomi e risulta particolarmente utile

quando si usano namespace diversi all'interno dello stesso documento ma uno di questi è quello più importante o il più utilizzato.

Può capitare di vedere che il nome del tag in cui viene definito il `nome` di namespace sia a sua volta preceduto dal `nome`:

```
<prefisso:tag xmlns:prefisso="URI">
```

ma ciò non cambia nulla ed è accettato dalla sintassi XML. Sebbene non ci sia da parte del parser XML nessun interesse nel contenuto dell'URI è buona norma che il contenuto dell'URI sia esplicativo del *significato* del namespace.

Pensandoci bene, per differenziare due nomi che potrebbero creare conflitto sarebbe bastato aggiungere un prefisso, senza bisogno dell'URI. L'URI viene però introdotta per avere allo stesso tempo un riferimento univoco e l'*indirizzo* di un *luogo* che possa contenere informazioni su quel namespace.

Quelli visti fino ad ora non sono gli unici modi per definire un nome di namespace, infatti può anche essere definito nel *prologo* nel modo seguente:

```
<?xml:namespace ns="URI"="prefisso">
```

La tecnica che noi utilizzeremo è la prima. Il motivo per cui sono importanti i namespace è il fatto che `rdf` sarà all'atto pratico un prefisso ossia un nome di namespace che permetterà di identificare in modo univoco i *nomi* riservati di RDF. In definitiva avremo che i namespace rappresentano dei repertori standard di nomi che permettono l'uniformità sintattica necessaria al funzionamento del Web semantico.

L'elemento vuoto

Talvolta capita che tutto quello che si vuole *dire* su di un elemento sia contenuto nel tag di apertura (i.e. quando tutta l'informazione è contenuta negli attributi).

Può quindi succedere che non sia necessario specificare nulla all'interno del valore dell'elemento. Per evitare di dover scrivere:

```
<nome att="val"></nome>
```

si introduce direttamente la barra rovesciata alla fine del tag di apertura ottenendo la scrittura equivalente:

```
<nome att="val"/>
```

che ha il grande pregio di essere molto più compatta, specie quando i nomi sono molto lunghi.

Questo risulterà molto utile per RDF XML in cui molto spesso capita che l'oggetto di una asserzione `(soggetto,proprietà,oggetto)` sia dato direttamente nel tag della `proprietà` attraverso l'attributo `rdf:resource`.

Considerazioni finali su XML

Uno dei vantaggi di XML è quello che, rappresentando i dati in un formato intermedio, permette di far *dialogare* tra loro applicativi che altrimenti non potrebbero farlo. Pur rappresentando un passo in avanti rispetto ad HTML, nel senso che l'autore di un documento XML può fornire delle informazioni esplicite sul contenuto del proprio documento, XML non è in grado di fornire di semantica i *suoi* tag. Vediamo attraverso un semplice esempio perché per il Web semantico dovremo spostarci verso altri linguaggi ed altre strutture per veicolare le nostre conoscenze sul *mondo*.

Quando si dice ad una persona qualcosa di nuovo, questa è in grado di *produrre* nuova informazione che è il frutto delle vecchie e delle nuove conoscenze. Se facciamo la stessa cosa con un computer *veicolando* l'informazione in XML questi sarà in grado

di inferire qualche cosa di nuovo solo se avremo *codificato* delle conoscenze aggiuntive in qualche programma che elabori quei specifici documenti XML; questa nuova conoscenza non nasce solo dall'informazione contenuta nel documento XML, serve qualche cosa di esterno.

Quello che si vuole fare è realizzare un *linguaggio* che permetta di codificare l'informazione, ma che al contempo contenga nella codifica stessa sufficienti informazioni e permetta di poter inferire nuova conoscenza. Sistemi diversi potranno *sfruttare* l'informazione per realizzare inferenze anche molto sofisticate, ma certe inferenze base dovranno sempre essere garantite da qualsiasi sistema.

Esempio 1.6.2. È noto che l'*essere genitore* è una relazione più generale dell'*essere mamma*. Sapendo che *Pia è la mamma di Franco* deve essere possibile inferire che *Pia è genitore di Franco*.

Questo tipo di inferenza deve però poter essere fatta attraverso la pura informazione contenuta nel *documento* e non deve utilizzare nulla di esterno. Ciò può essere fatto sfruttando la semantica della proprietà *subProperty* ed introducendo una asserzione che dice che l'*essere mamma* è una *subProperty* dell'*essere genitore*. XML non potrebbe mai fare nulla del genere direttamente. Per farlo si dovrebbe costruire un programma C++ che definisca in sé la semantica del tag *subProperty*.

Un altro problema che affligge XML e che si vorrebbe risolvere nel Web semantico è quello della non unicità dei nomi. Succede infatti che due persone diverse definiscano due tag diversi che però indichino lo stesso oggetto. In XML non esiste nessun meccanismo esplicito per permettere di *dire* che i due nomi sono in realtà lo stesso nome e quindi comportarsi di conseguenza.

Vedremo come DAML+OIL risponda a queste ed altre esigenze fornendo meccanismi espressivi sufficientemente sofisticati per permettere di realizzare i primi passi verso la realizzazione del Web semantico. Per poter parlare di DAML+OIL dobbiamo però prima affrontare RDF ed RDF Schema Language che, assieme ad XML, forniranno gli strumenti necessari per *costruire* DAML+OIL.

1.7 Resource Description Framework (RDF)

Abbiamo visto che XML non è sufficiente per *fornire* di semantica i contenuti di un documento ed abbiamo già accennato al fatto che l'idea alla base del Web semantico è quella di realizzare delle *reti semantiche* connesse tra loro. Dovrebbe essere altresì chiaro a questo punto il meccanismo dei namespace di XML, la tecnica di *nominare* oggetti attraverso le URI e cosa significhi *marcare* un testo. Quello che serve a questo punto è un linguaggio che permetta di poter *costruire* le *triple* `(soggetto,predicato,oggetto)` che andranno a *realizzare* la rete semantica che è il supporto fondamentale al Web semantico.

Il Resource Description Framework (RDF) è il primo passo, dopo XML, compiuto dal W3C in questa direzione. L'idea base di RDF è quella di utilizzare dei *metadati* per descrivere i *dati* contenuti in una pagina. RDF sarà quindi utilizzato per aggiungere *metainformazioni* ad ogni *risorsa* identificata da una URI. Tali metainformazioni possono essere *semplici* (e.g. l'autore di una risorsa è una certa altra risorsa) o *complesse* (e.g. quando ci si riferisce ad una intera ontologia).

RDF, rifacendosi al principio del *least power*, metterà a disposizione gli *strumenti* minimi per *costruire* asserzioni e citazioni (asserzioni su asserzioni). Va però sottolineato fin dall'inizio che anche se RDF serve per rappresentare la conoscenza non ha

in sé nessun meccanismo di deduzione automatica.

Eventuali inferenze si potranno fare solo avvalendosi di una semantica esplicita e di *motori inferenziali* esterni. Chiaramente, il secondo aspetto presuppone il primo, ma allo stato attuale mancano entrambi (in forma standardizzata). La constatazione di questa mancanza è stata una delle motivazioni per la forma attuale di questa tesi. Il primo passo, chiaramente, è la definizione di una semantica che descriva le conseguenze di un insieme di asserzioni RDF. In altre parole, solo la sintassi è stata finora standardizzata. La semantica, e quindi l'implicazione, sono a livello di *draft*, il quale draft risulta di difficile accessibilità vis-a-vis la semantica standard di DATALOG (Logic Based Data Model) in cui già si può catturare il linguaggio delle reti semantiche. In questa direzione di lavoro, la proposta più simile alla nostra è quella attualmente proposta dal gruppo di R. Fikes, D. McGuinness e S. McIlraith ai KSL [3] (Knowledge Systems Laboratory) della Stanford University.

Lo scopo di RDF è quello di fornire una *cornice* espressiva per rappresentare informazioni che possano così essere scambiate tra applicazioni diverse senza perdita di *significato*. Il rendere il formato della rappresentazione indipendente dal *programma applicativo* permetterà lo *scambio* e la *condivisione* di conoscenza anche in contesti per i quali i dati non erano stati originariamente concepiti.

1.7.1 Il modello RDF

RDF è un modello di descrizione astratto e non pone vincoli sulla sintassi e sul significato delle descrizioni di una risorsa. Questo vuol dire che ognuno potrebbe proporre un qualsiasi meccanismo per descrivere risorse utilizzando le astrazioni previste da RDF. Ad esempio, una asserzione RDF potrebbe essere espressa tramite una

rappresentazione grafica.

Tuttavia, per ragioni pratiche, la definizione formulata del W3C propone XML come metalinguaggio per la rappresentazione del modello RDF, cioè propone RDF come linguaggio basato su XML. Questo modello di descrizione si basa su tre tipi di oggetti:

Definizione 1.7.1. (Risorsa)
È una qualsiasi cosa che possa essere nominata.

Commento 1.7.1. Una risorsa è sempre identificata da una URI secondo il meccanismo già descritto.

Esempio 1.7.1. Qualsiasi cosa può essere identificato da una URI, anche oggetti che fanno parte del dominio materiale come ad esempio un libro in uno scaffale, ma anche proprietà quali l'*avere come autore*.

Definizione 1.7.2. (Proprietà)
Una *proprietà* consiste in un aspetto specifico, una caratteristica, un attributo, o una relazione usata per descrivere una risorsa. Ogni proprietà ha un significato specifico, definisce i valori ammessi, i tipi di risorse a cui può riferirsi, e la sua relazione con altre proprietà.

Commento 1.7.2. Una proprietà può essere vista come una risorsa e quindi identificata da una URI.

Esempio 1.7.2. La proprietà *avere come autore* avrà un suo dominio, i *libri*, ed un suo codominio, le *persone*.

Definizione 1.7.3. (Asserzione)
Una *asserzione* RDF è una *frase* con un *soggetto*, una risorsa, un *verbo*, una proprietà,

ed un *complemento*, una risorsa o un letterale (stringa di caratteri). Una asserzione è quindi una tripla (soggetto, verbo, complemento). In RDF le tre componenti della tripla sono dette `(subject, predicate, object)`.

Commento 1.7.3. In termini RDF un letterale può comprendere nel suo contenuto anche mark-up XML, ma esso non è elaborato da un processore RDF. Esistono alcune restrizioni sintattiche su come può essere espresso il mark-up nei letterali.

Esempio 1.7.3. La frase della lingua italiana *Franco è l'autore della tesi sul Web semantico* per essere rifrasata in una asserzione RDF deve essere *trasformata* in una tripla `(subject,predicate,object)`, dove `subject` e `predicate` devono essere obbligatoriamente delle risorse mentre `object` può essere sia una risorsa che un letterale. Per fare ciò associamo la URI `http://example.org/Tesi` all'oggetto *tesi sul Web semantico* che appartiene al dominio del discorso, la URI `http://example.org/autore` alla proprietà *avere come autore* anch'essa facente parte del dominio del discorso e il letterale `Franco` al nome Franco che interpretato identifica me stesso. Così facendo otteniamo l'asserzione RDF:

```
(http://example.org/Tesi,http://example.org/autore,Franco)
```

Definizione 1.7.4. (Grafo RDF)
Un grafo RDF è un grafo orientato in cui il vertice di partenza di ogni arco rappresenta il `subject`, l'arco stesso il `predicate` e il nodo di arrivo l'`object` di una asserzione RDF.

Commento 1.7.4. Un nodo *ovale* identifica una risorsa mentre uno *rettangolare* un letterale.

Esempio 1.7.4. Data l'asserzione RDF:

`(http://example.org/Tesi,http://example.org/autore,Franco)`

che traduce la frase *Franco è l'autore della tesi sul Web semantico* possiamo rappresentarla con il più intuitivo grafo di figura 1.2 a pagina 57.

Figura 1.2: *La tesi sul Web semantico ha come autore Franco.*

Nell'esempio 1.7.4 abbiamo visto che l'oggetto dell'asserzione era il letterale `Franco` che tutto sommato non ha in sé nessun significato esplicito. Possiamo quindi pensare di raffinare l'esempio precedente.

Esempio 1.7.5. Pensiamo ora di voler *descrivere* Franco in modo più dettagliato ad esempio asserendo che ha una e-mail ed un nome. Per fare ciò definiamo una nuova risorsa `http://example.org/Franco` che sarà il *nome* di Franco, e su questa usandola come soggetto, facciamo le debite asserzioni. Per fare le asserzioni volute introduciamo anche le due nuove proprietà `http://example.org/nome` e `http://example.org/e-mail`. Quello che si ottiene è il grafo di figura 1.3 a pagina 58.

Il vantaggio di usare una URI per nominare una proprietà, al posto di usare semplicemente un letterale quale `autore` , ha dei grandi vantaggi:

- la proprietà può essere descritta dettagliatamente all'interno della *pagina* identificata dalla URI stessa utilizzando dei meccanismi o verbali o più strutturati come vedremo in seguito, permettendo così di fornirla di semantica;
- una volta *depositata* all'interno della pagina identificata dalla sua URI quella

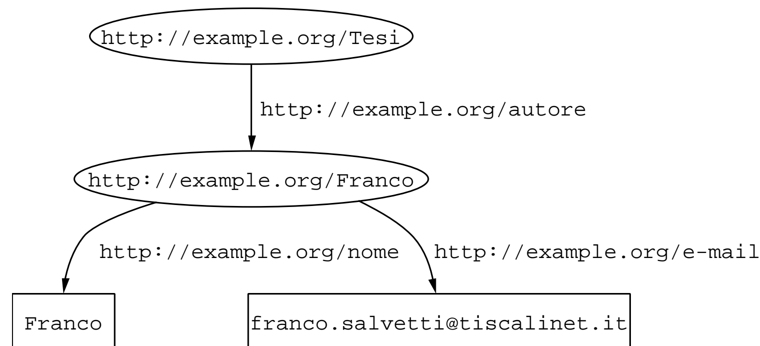


Figura 1.3: *Franco* ora è una risorsa.

proprietà potrà essere usata da chiunque, in qualsiasi pagina del Web semantico per descrivere una qualsiasi cosa, mantenendone però il significato originale;

- il fatto di usare una URI risolve, come già spiegato, il problema dei conflitti sui nomi;
- avendo *trasformato* un letterale in una risorsa sarà possibile trattarla come tale e quindi utilizzarla come soggetto o oggetto in una asserzione RDF;
- utilizzare delle URI per nominare soggetto, oggetto e predicato permetterà di sviluppare e condividere dei *vocabolari* che esprimano una conoscenza pubblica e condivisa sul Web semantico.

Esempio 1.7.6. Mostriamo ora come le potenzialità messe in luce possano effettivamente funzionare. Come esempio mostriamo la proprietà nominata tramite la URI:

`http://dublincore.org/2001/08/14/dces#creator`

Se andiamo a vedere il contenuto della URL uguale alla stessa URI, troveremo un documento RDF che descrive in modo non ambiguo che `creator` è una proprietà che serve per identificate i *creatori* di *risorse*:

> "*An entity primarily responsible for making the content of the resource. Examples of a Creator include a person, an organisation, or a service. Typically, the name of a Creator should be used to indicate the entity*".

Quanto riportato sopra in modo verbale è il `comment` della definizione formale della proprietà stessa. Questa proprietà fa parte del vocabolario Dublin Core che è stato sviluppato per descrive risorse all'interno del dominio semantico delle biblioteche. Quindi chiunque utilizzi la URI che abbiamo descritto prima come predicato per scrivere una asserzione RDF lo farebbe in un modo non ambiguo. Tutti condividono la stessa definizione di `creator` data all'interno del namespace del Dublin Core.

Sarebbe possibile scrivere un programma che quando trova una asserzione che utilizza come predicato la URI relativa a `creator` la interpreti confidando che l'oggetto dell'asserzione RDF è sicuramente l'identificativo (letterale o URI) di un *creatore*. Un motore di ricerca potrebbe ad esempio cercare tutte le asserzioni che hanno come predicato la URI relativa a `creator` e come oggetto la URI `http://example.org/Franco` ritornando tutti i `subject`, ossia tutti gli identificatici di risorse, di cui io sono il *creatore* e di cui qualcuno ha parlato sul Web semantico. In questo modo si eviterebbe di ricevere come risposta all'interrogazione pagine in cui compare il mio identificativo ma relativo ad asserzioni che nulla hanno a che vedere con la mia veste di *creatore*.

Vedremo in seguito come sia possibile descrivere le proprietà nominate tramite delle URI attraverso delle asserzioni RDF. Per riassumere e capire la capacità espressiva di RDF diciamo che nel Web semantico:

"*Qualsiasi cosa può dire qualsiasi cosa su qualsiasi cosa*". – Aaron Swartz

Abbiamo visto che una stessa asserzione RDF può essere rappresentata in modi diversi. Ciò conduce naturalmente alla definizione 1.7.5.

Definizione 1.7.5. (Equivalenza tra asserzioni RDF)
Un modello RDF è un modo indipendente dalla sintassi di rappresentare una asserzione RDF. Diremo infatti che due asserzioni RDF sono *equivalenti* se hanno lo stesso modello.

Data la definizione 1.7.5 vediamo ora alcuni modi equivalenti di rappresentare una asserzione RDF. L'utilità di avere modi diversi, ma equivalenti, di rappresentare una asserzione RDF risulterà chiaro nel prosieguo della trattazione.

1.7.2 Rappresentazioni equivalenti del modello RDF

Abbiamo già visto due modi equivalenti di rappresentare una asserzione RDF. Il primo è esplicitandone la tripla `(subject, predicate, object)`, mentre il secondo è usandone il grafo definito in 1.7.4. Vediamo ora alcuni modi alternativi di cui quello che utilizza SMODELS è qui presentato per la prima volta.

N-Triples

N-Triples, Notation 3 o N3 sono linguaggi *orientati* alle triple introdotti per rappresentare asserzioni RDF. Un documento scritto in N-Triples può essere visto semplicemente come una sequenza di asserzioni dove ogni asserzione è della forma (soggetto, predicato, oggetto) espresse come:

```
<soggetto> <predicato> <oggetto>.
```

Come abbiamo già detto una URI può servire per nominare qualsiasi cosa, per cui tutte le tre parti di una asserzione RDF, essendo delle risorse, possono essere delle URI. Pensiamo quindi di associare ad ogni parte delle frase *A ama B* una URI. Quello che si può ottenere è una asserzione RDF che espressa nel linguaggio N-Triples sarà:

```
<http://example.org/A><http://example.org/ama><http://example.org/B>.
```

Il linguaggio N-Triples ha una sua grammatica molto più estesa di quanto visto fino ad ora, ma noi siamo interessati ad RDF e quindi utilizzeremo N-Triples solo per esprimere asserzioni RDF.

Prolog

Una asserzione RDF può essere vista come un *fatto* Prolog [2]. Avremo quindi che l'asserzione RDF relativa alla frase *A ama B* può essere tradotta come in un programma Prolog. Una possibilità è quella di lasciare il predicato `ama` esplicito:

```
%predicate(subject,object)

'http://example.org/ama/'('http://example.org/A/',
                          'http://example.org/B/').
```

un altra è quella di *reificare*:

```
%triple(subject,predicate,object)

triple('http://example.org/A/',
       'http://example.org/ama/',
       'http://example.org/B/').
```

Smodels

In modo analogo si può procedere per SMODELS, ottenendo:

```
%predicate(subject,object)

"http://example.org/ama/"("http://example.org/A/",
                          "http://example.org/B/").
```

o alternativamente in versione *reificata*:

```
%triple(subject,predicate,object)

triple("http://example.org/A/",
       "http://example.org/ama/",
       "http://example.org/B/").
```

Quest'ultima rappresentazione sarà quella che utilizzeremo quando daremo la semantica per traduzione delle più importanti proprietà DAML+OIL.

1.7.3 Sintassi RDF XML

Il modo più comune per scrivere delle asserzioni RDF è utilizzando XML. In considerazione che normalmente più asserzioni sulla stessa risorsa sono fatte contemporaneamente la sintassi RDF XML permette di raggrupparle all'interno dello stesso tag `rdf:Description`. Osserviamo che `rdf:Description` è un nome all'interno del *namespace* `rdf` che a sua volta altro non sarà che una URL `presso` la quale si troverà un documento che *definisce* i nomi usati da RDF.

Per poter fare una asserzione RDF prima di tutto dobbiamo identificare il `subject`; ci sono due modi per farlo. Il primo è assegnando il *nome* della risorsa che vogliamo descrivere all'attributo `rdf:about` del tag `rdf:Description` (i.e. l'URI che identifica la risorsa). L'altro è quello di assegnare un letterale all'attributo `rdf:ID` del tag

`rdf:Description`. In quest'ultimo caso il nome specificato diventa il nome *locale* di una risorsa per la quale non esiste ancora una URI.

L'attributo `rdf:ID` indica la creazione di una nuova risorsa mentre l'attributo `rdf:about` serve per riferirsi ad una risorsa già esistente; quindi, in `rdf:Description` è possibile specificare o `rdf:ID` o `rdf:about`, ma non entrambi contemporaneamente. I valori per ogni attributo `rdf:ID` non devono apparire in più di un attributo `rdf:ID` all'interno dello stesso documento (i.e. non si può *creare* più volte la stessa risorsa).

Quando nel tag `rdf:Description` viene specificato l'attributo `rdf:about` tutte le asserzioni contenute in `rdf:Description` si riferiscono alla risorsa il cui identificatore è indicato da `rdf:about`. Il valore dell'attributo `rdf:about` è interpretato come un riferimento ad URI. Un tag `rdf:Description` senza nessun attributo specificato rappresenta una nuova risorsa. Tale risorsa potrebbe essere un surrogato, o proxy, per qualche altra risorsa fisica che non abbia un URI riconoscibile (i.e. una risorsa *fittizia*). Il valore dell'attributo `rdf:ID` del tag `rdf:Description`, se presente, rappresenta l'*anchor id* ad una nuova risorsa `online`. Se un altro `rdf:Description` o un `object` deve essere riferito alla risorsa `online`, questo userà il valore dell'`rdf:ID` preceduto dal carattere `#`.

All'interno di un tag `rdf:Description` si trovano le asserzioni vere e proprie che hanno come soggetto quello specificato negli attributi di `rdf:Desctriprion`. La struttura per fare queste asserzioni è:

```
<predicate>object</predicate>
```

Se l'`object` non è un letterale ma una URI questa va inserita direttamente nel tag `predicate` attraverso l'attributo `rdf:resource`.

Esempio 1.7.7. Pensiamo di aver dichiarato il namespace `example` e di voler scrivere l'asserzione RDF *Franco è l'autore della tesi sul Web semantico* usando la sintassi XML. Se pensiamo a `Franco` come ad un letterale quello che dovremo scrivere è:

```
<rdf:Description rdf:about="http://example/Tesi">
  <example:autore>
     Franco
  </example:autore>
</rdf:Description>
```

Mentre se pensiamo a `Franco` come ad una risorsa scriveremo:

```
<rdf:Description rdf:about="http://example/Tesi">
  <example:autore rdf:resource="http://example/Franco"/>
</rdf:Description>
```

Si potrebbe pensare che RDF sia una istanza di XML, ossia un linguaggio definito da XML attraverso un DTD. Ma non esiste un DTD di XML che descrive RDF. La definizione di come si scrive una asserzione RDF in XML è data esplicitamente attraverso una grammatica EBNF (Extended Backus-Naur Form). Avremo quindi che la validazione di un documento XML che codifica asserzioni RDF non si basa su di un DTD ma sulla grammatica stessa di RDF XML.

Vediamo qui di seguito la versione semplificata della grammatica RDF XML a cui ci riferiremo per costruire le asserzioni RDF espresse tramite XML.

```
RDF             ::= ['<rdf:RDF>'] description* ['</rdf:RDF>']
description     ::= '<rdf:Description' idAboutAttr? '>' propertyElt*
                    '</rdf:Description>'
idAboutAttr     ::= idAttr | aboutAttr
aboutAttr       ::= 'about="' URI-reference '"'
idAttr          ::= 'ID="' IDsymbol '"'
propertyElt     ::= '<' propName '>' value '</' propName '>'
```

```
                  | '<' propName resourceAttr '/>'
propName        ::= Qname
value           ::= description | string
resourceAttr    ::= 'resource="' URI-reference '"'
Qname           ::= [ NSprefix ':' ] name
URI-reference   ::= string, interpreted per [URI]
IDsymbol        ::= (any legal XML name symbol)
name            ::= (any legal XML name symbol)
NSprefix        ::= (any legal XML namespace prefix)
string          ::= (any XML text, with "<", ">", and "&" escaped)
```

L'elemento radice di un documento RDF XML è sempre `<rdf:RDF>` e una asserzione è sempre fatta attraverso un tag `<rdf:Descriprion>`.

Esempio 1.7.8. Vediamo ora come è fatto un intero documento RDF XML che *modellizzi* la frase *A ama B.*

```
<?xml version="1.0"?>
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:example="http://example.org/">
 <rdf:Description rdf:about="http://example.org/A">
  <example:ama rdf:resource="http://example.org/B"/>
 </rdf:Description>
</rdf:RDF>
```

Di cui la relativa rappresentazione attraverso un grafo orientato risulta essere come in figura 1.4 a pagina 65.

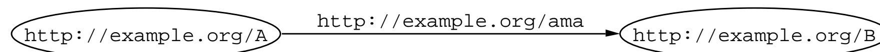


Figura 1.4: Una semplice asserzione RDF [esempio: 1.7.8].

Mentre la rappresentazione tramite N-Triples è la seguente:

```
<http://example.org/A><http://example.org/ama><http://example.org/B>.
```

Dove abbiamo che `http://example.org/A/` è il `subject`, `http://example.org/ama` è il `predicate` e `http://example.org/B/` è l'`object`.

Probabilmente alcune parti del documento RDF XML risultano ancora oscure, per questo vediamo un ulteriore esempio che spieghi definitivamente come viene costruito un documento RDF XML.

Esempio 1.7.9. Pensiamo di voler scrivere la frase *Franco è l'autore della tesi sul Web semantico* come una asserzione RDF XML. Prima di tutto diciamo che i nomi delle proprietà per descrivere delle risorse *devono* essere associati ad un namespace (i.e. un documento all'interno del quale vengono definite proprietà e loro semantica). Per poter utilizzare una proprietà che appartiene ad un dato namespace si deve dichiarare un `prefisso` ed utilizzarlo come prefisso del nome della proprietà. In questo tipo di documenti sono contenute informazioni di varia natura sulle proprietà ed in generale costituiscono una ontologia. Vedremo poi quando introdurremo RDF Schema Language come si costruiscono le ontologie, per il momento pensiamo che ne esistano di disponibili.

Vediamo quindi come potrebbe essere fatto un documento XML che codifica lo statement RDF dato in precedenza. Prima di tutto si deve dire che quel documento è un documento XML:

```
<?xml version="1.0"?>
```

a questo punto si crea un *wrapper* per dire che quello che sarà lì contenuto è uno modello di dati RDF:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
```

```
</rdf:RDF>
```

Osserviamo che abbiamo dichiarato all'interno del tag RDF il prefisso `rdf` facendolo riferire ad una URI alla quale si può trovare un l'ontologia relativa ad RDF stesso. Questa ontologia unitamente alla grammatica EBNF che abbiamo visto a pagina 64 definisce RDF. Vedremo poi che all'interno dell'ontologia RDF vi siano riferimenti anche ad altre ontologie, ma per il momento non ce ne preoccupiamo.

Torniamo al nostro problema iniziale, ossia quello di scrivere un semplice asserzione RDF attraverso un modello RDF scritto nella sintassi XML. A questo punto dichiariamo anche un prefisso per riferirci ad una ontologia che contenga i nomi delle proprietà che consideriamo lecite ed utili nel nostro *dominio*:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http://example.org/onto#">
</rdf:RDF>
```

Adesso siamo pronti per scrivere l'asserzione vera e propria, per farlo utilizziamo il tag `rdf:Description` che appartiene allo spazio dei nomi `rdf` in cui attraverso l'attributo `rdf:about` dichiariamo il soggetto dello statement ossia la risorsa che vogliamo descrivere, fatto ciò all'interno del tag `ont:autore` (la proprietà), che appartiene allo spazio dei nomi della nostra personale ontologia, attraverso un letterale ne diamo il valore:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http://example.org/">
<rdf:Description rdf:about="http://example.org/Tesi">
  <ont:autore>Franco</ont:autore>
</rdf:Description>
</rdf:RDF>
```

A cui corrisponde il grafo in figura 1.2 a pagina 57.

Esempio 1.7.10. Analogamente a quanto già fatto precedentemente mostriamo qui il documento RDF XML nel caso in cui `Franco` non fosse un letterale ma una risorsa con un nome ed una e-mail.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http://example.org/">
<rdf:Description rdf:about="http://example.org/Tesi">
   <ont:autore rdf:resource="http://example.org/Franco"/>
</rdf:Description>
<rdf:Description rdf:about="http://example.org/Franco">
   <ont:Nome>Franco</ont:Nome>
   <ont:e-mail>franco.salvetti@tiscalinet.it</ont:e-mail>
</rdf:Description>
</rdf:RDF>
```

A cui corrisponde il grafo in figura 1.3 a pagina 58.

1.7.4 Sintassi abbreviata RDF XML

La sintassi RDF XML vista fino ad ora è detta sintassi *serializzata.* Anche se ciò non introduce nulla di nuovo vediamo una sintassi alternativa più compatta che però è talvolta utilizzata in documenti RDF reperibili sul Web semantico.

Se il nome di una proprietà (e.g. `ont:autore`) compare una sola volta all'interno di un tag `rdf:Description` ed ha come `object` un letterale (e.g. `Franco`) questo può comparire direttamente all'interno del tag `rdf:Description` come un attributo.

Esempio 1.7.11. Volendo asserire che *Franco è l'autore della tesi sul Web semantico* potremmo scrivere, in modo alternativo a quello visto nell'esempio 1.7.9:

```
<rdf:Description
 rdf:about="http://example.org/Tesi"
 ont:autore="Franco">
```

Quando vi siano dei tag `rdf:Description` *logicamente nidificati* si può optare per una scrittura alternativa nella quale la *descrizione* della risorsa che rappresenta il valore (e.g. `http://example.org/Franco`) di una certa proprietà (e.g `ont:autore`) è data direttamente e non *successivamente*.

Esempio 1.7.12. Volendo asserire che `http://example.org/Franco`, *l'autore della tesi sul Web semantico*, ha un nome ed una e-mail potremmo scrivere, in modo alternativo a quello visto nell'esempio 1.7.10:

```
<rdf:Description rdf:about="http://example.org/Tesi">
   <ont:autore>
      <rdf:Description rdf:about="http://example.org/Franco">
         <ont:Nome>Franco</ont:Nome>
         <ont:e-mail>franco.salvetti@tiscalinet.it</ont:e-mail>
      </rdf:Description>
   </ont:autore>
</rdf:Description>
```

Sfruttando il fatto che la descrizione della risorsa `http://example.org/Franco` è fatta soltanto di proprietà che compaiono una sola volta e che hanno valori letterali, si può riscrivere come:

```
<rdf:Description rdf:about="http://example.org/Tesi">
   <ont:autore>
      <rdf:Description rdf:about="http://example.org/Franco"
         ont:Nome="Franco"
         ont:e-mail="franco.salvetti@tiscalinet.it"/>
   </s:autore>
</rdf:Description>
```

1.7.5 I contenitori

Talvolta capita di voler descrivere una risorsa attraverso una proprietà che assume più valori (simultanei o non simultanei). Ad esempio si può voler dire che un certo libro è stato scritto da un certo *insieme* di autori o che i dessert di una certa cena potranno essere budino o torta (ma non entrambi).

Per fare ciò RDF XML mette a disposizione il concetto di *contenitore* attraverso la *classe*[4] `rdf:Container` e le tre sue sottoclassi `rdf:Bag`, `rdf:Seq` e `rdf:Alt`.

- La classe `rdf:Bag` è una sorta di *multi-insieme* di risorse o letterali in cui cioè non esiste un ordine ed è ammessa la duplicazione dei nomi.
- La classe `rdf:Seq` *rappresenta* una sequenza ordinata di risorse o letterali.
- La classe `rdf:Alt` *rappresenta* una lista di alternative possibili (la proprietà deve assumere uno ed uno solo dei valori specificati nella lista dei valori possibili).

In RDF una proprietà assume come valore una risorsa identificata tramite una URI. Per riuscire a mantenere questa struttura con i contenitori si deve introdurre una risorsa *fittizia* che rappresenterà una istanza di una delle sottoclassi di `rdf:Container`, il contenitore appunto. Fatto ciò, questa risorsa - il contenitore - andrà descritta attraverso delle *normali* asserzioni RDF.

La prima operazione dare fare è quindi quella di creare il contenitore. Ciò è fatto tramite la proprietà `rdf:type` che assumerà come valore `rdf:Bag` o `rdf:Seq` o `rdf:Alt`. Successivamente, per *riempire* il contenitore, si dovranno fare un certo numero di asserzioni RDF che abbiano come `subject` la risorsa fittizia che rappresenta

[4] Il concetto di classe verrà presentato durante la trattazione di RDF Schema Language. Per il momento senza perdere di generalità si può pensare ad una classe nell'ambito RDF rifacendosi all'usuale concetto di classe dei linguaggi object oriented.

il contenitore. Per collegare i contenuti al contenitore vengono introdotte surrettiziamente delle proprietà di nome `_1 _2` [...] attraverso la proprietà `rdf:li`. Queste proprietà sono istanze della classe `rdfs:ContainerMembershipProperty`, sottoproprietà della proprietà `rdfs:member` e servono quindi per collegare il contenitore a tutte le risorse che esso contiene.

Formalmente la grammatica EBNF per i contenitori è la seguente:

```
container       ::= sequence | bag | alternative
sequence        ::= '<rdf:Seq' idAttr? '>' member* '</rdf:Seq>'
bag             ::= '<rdf:Bag' idAttr? '>' member* '</rdf:Bag>'
alternative     ::= '<rdf:Alt' idAttr? '>' member+ '</rdf:Alt>'
member          ::= referencedItem | inlineItem
referencedItem  ::= '<rdf:li' resourceAttr '/>'
inlineItem      ::= '<rdf:li>' value '</rdf:li>'
```

I contenitori possono essere utilizzati ovunque si possa utilizzare `rdf:Description`. Ciò impone di estendere la grammatica EBNF data a pagina 64 includendo anche:

```
RDF             ::= '<rdf:RDF>' obj* '</rdf:RDF>'
value           ::= obj | string
obj             ::= description | container
```

Esempio 1.7.13. Vediamo ora il documento RDF XML che rappresenta la frase *I prerequisiti necessari per l'ammissione a Stanford sono TOEFL e GRE* utilizzando il concetto di contenitore.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http://example.org/onto#">
<rdf:Description rdf:about="http://example.org/Stanford">
   <ont:requirements>
      <rdf:Bag>
```

```
        <rdf:li rdf:resource="http://www.toefl.org"/>
        <rdf:li rdf:resource="http://www.gre.org"/>
      </rdf:Bag>
   </ont:requirements>
</rdf:Description>
</rdf:RDF>
```

In figura 1.5 a pagina 72 si può osservare come il nodo `genid:27855` rappresenti la risorsa fittizia che è il *contenitore* di tipo `rdf:Bag` e di come siano state introdotte implicitamente dalla proprietà `rdf:li` le proprietà `_1` e `_2`. Si osserva che, sempre implicitamente, viene introdotto il nodo che rappresenta la risorsa `rdf:Bag` collegata da un arco etichettato dalla proprietà `rdf:type` al nodo fittizio, ciò è fatto per rendere esplicito il fatto che il nodo fittizio; sia effettivamente un contenitore di tipo `rdf:Bag`.

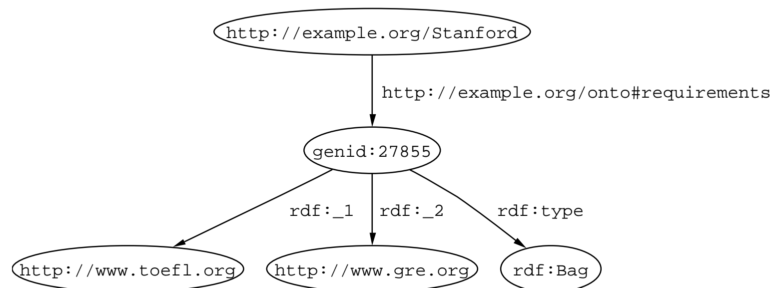


Figura 1.5: Un contenitore RDF [esempio: 1.7.13].

Asserzioni sugli elementi di un contenitore

È possibile *costruire* un contenitore dandogli un nome attraverso l'attributo `rdf:ID` nel tag `rdf:Descriprion`. In questo modo, essendo una risorsa con nome, si potranno fare delle asserzioni RDF su quel contenitore usando il riferimento `#nome` dove `nome` è la stringa di caratteri assegnata all'attributo `rdf:ID`.

Se però facciamo una asserzione sulla risorsa `#nome` stiamo descrivendo il contenitore o gli elementi contenuti? Per ovviare a questo problema viene introdotto l'attributo `rdf:aboutEach` per il tag `rdf:Description`. Se si utilizza l'attributo `rdf:aboutEach=#nome` in una asserzione RDF, stiamo descrivendo gli elementi contenuti nel contenitore di nome `nome` e non il contenitore stesso (cosa che possiamo sempre fare usando l'attributo `rdf:about`).

Ovviamente, se stiamo usando l'attributo `rdf:aboutEach`, visto che stiamo descrivendo ciò che è contenuto nel contenitore, stiamo implicitamente descrivendo solo i contenuti che sono risorse; non possiamo descrivere i letterali.

Esempio 1.7.14. Mostriamo il documento RDF XML che modellizza la frase *Franco ed Alessandra, tesisti del Prof. Provetti, studiano* SMODELS.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http:/example.org/onto#">
<rdf:Bag rdf:ID="TesistiProfProvetti">
   <rdf:li rdf:resource="http://example.org/Franco"/>
   <rdf:li rdf:resource="http://example.org/Alessandra"/>
</rdf:Bag>
<rdf:Description rdf:aboutEach="#TesistiProfProvetti">
   <ont:studia rdf:resource="http://example.org/smodels"/>
</rdf:Description>
</rdf:RDF>
```

In figura 1.6 a pagina 74 si può osservare come tutte le risorse contenute nell'`rdf:Bag` di nome `online:#TesistiProfProvetti` siano state effettivamente descritte.

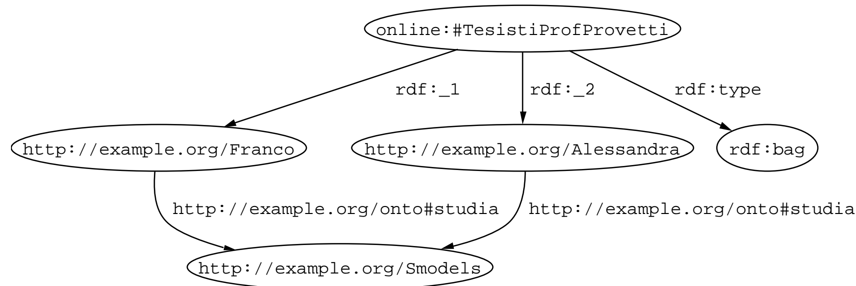


Figura 1.6: Descrizione delle risorse contenute in un `rdf:Bag` [esempio: 1.7.14].

1.7.6 Asserzioni su asserzioni

Come abbiamo già accennato a pagina 16 in RDF è possibile fare asserzioni, ma anche asserzioni su asserzioni (i.e. citazioni). Questo meccanismo corrisponde alla reificazione di una asserzione, ossia alla sua *trasformazione* in una risorsa. Ciò risulta utile perché permette di *incapsulare* asserzioni all'interno di altre asserzioni. Vediamo come se deve procedere attraverso un esempio:

Esempio 1.7.15. Si pensi di voler tradurre la *citazione* `Alessandro Provetti ha detto che Franco è l'autore della tesi sul Web semantico` in una asserzione RDF XML.

Possiamo osservare che all'interno della *citazione* vi sia la frase *Franco è l'autore della tesi sul Web semantico*. Per riportare la citazione ad essere una asserzione dovremo *rendere* la frase *Franco è l'autore della tesi sul Web semantico* come una risorsa. Per fare ciò si introduce una nuova risorsa attraverso l'attributo `rdf:ID`, che rappresenterà la *frase*, e si faranno su di essa quattro asserzioni. La prima dirà che quella risorsa è una asserzione; ciò si ottiene dicendo che la risorsa appena definita è istanza della classe `rdf:Statement`. Come abbiamo già visto per *istanziare*

oggetti da una classe si usa la proprietà `rdf:type`. Si procede poi dicendo che quel-l'`rdf:Statement` ha un soggetto (`rdf:subject`) un predicato (`rdf:predicate`) ed un oggetto (`rdf:object`) che saranno rispettivamente le risorse che individuano la *Tesi*, l'*avere autore* e *Franco*. Formalmente in RDF XML si ottiene:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http://example.org/onto#">
<rdf:Description rdf:ID="Frase">
 <rdf:type
 rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
 <rdf:subject
 rdf:resource="http://example.org/Tesi"/>
 <rdf:predicate
 rdf:resource="http://example.org/autore/"/>
 <rdf:object
 rdf:resource="http://example.org/Franco"/>
</rdf:Description>
<rdf:Description rdf:about="#Frase">
 <ont:asseritaDa rdf:resource="http://example.org/Provetti"/>
</rdf:Description>
</rdf:RDF>
```

Si può osservare, sia nel documento RDF XML che nella figura 1.7 a pagina 76, che l'aver introdotto la proprietà `ont:asseritaDa` ci ha permesso di *asserire* che la risorsa `online:#Frase` è stata *asserita* dalla risorsa `http://example.org/Provetti`.

1.8 RDF Schema Language

Fino a questo punto abbiamo parlato del fatto che RDF mette a disposizione un linguaggio semplice e di uso generale per rappresentare la conoscenza che abbiamo sul

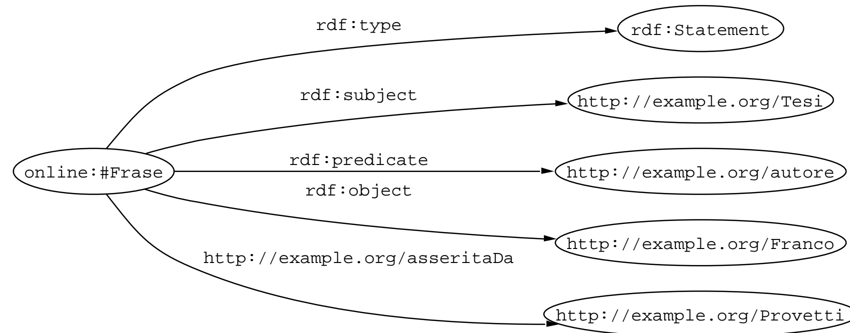


Figura 1.7: Asserzioni su asserzioni in RDF [esempio: 1.7.15].

mondo. Abbiamo visto che esprimiamo questa conoscenza, almeno in prima istanza, in termini di triple di risorse `<subject> <predicate> <object>`. Ma come può *capire* un computer che cosa significhi un certa proprietà con cui abbiamo descritto una certa risorsa? E come è possibile *creare* nuove proprietà? Nell'esempio 1.7.8 abbiamo asserito che `<A> <example:ama> <B>`, e lo abbiamo fatto usando una URI per identificare la risorsa che rappresenta la proprietà *ama*, ma questa URI è inutile se non ne esplicitiamo il *significato*.

È qui che entrano in gioco gli Schemi (i.e. ontologie) che sono *modi* per descrivere il significato di relazioni e classi. Questa descrizione aiuterà i computer ad utilizzare queste relazioni in modo più efficace (e.g. *capendo* come nomi diversi rappresentino lo stesso oggetto).

Quello che serve ora è un linguaggio che metta a disposizione gli strumenti minimi per poter *costruire* delle ontologie. Per farlo introdurremo RDF Schema Language che è un linguaggio che estende RDF fornendo un insieme di strumenti linguistici che permettono di realizzare nuove *classi* e nuove *proprietà*, e di descrivere le relazioni

tra esse. È inteso che le ontologie (i.e. i vocabolari del Web semantico) sono strutturate in modo gerarchico in termini di classi e di proprietà, permettendo quindi di derivare per ereditarietà nuova conoscenza. Sarà infatti possibile definire la classe `ont:Animale` come istanza della classe generale `rdfs:Class` e da essa derivare la sottoclasse `ont:Cane` attraverso la proprietà `rdfs:subClassOf`. A questo punto se `Pippo` è istanza della classe `ont:Cane` potremmo inferire automaticamente (i.e. senza avere coscienza sul reale significato dei nomi delle classi) che `Pippo` è un istanza della classe `ont:Animale`.

Questi vocabolari *risiederanno* in opportuni documenti reperibili direttamente sul Web semantico, ai quali sia agenti software che esseri umani potranno accedere. Per usare una proprietà definita in una certa ontologia basterà usare il suo nome, dove il suo nome sarà tipicamente la URL della pagina Web nella quale risiede, seguita dal carattere `#` seguito a sua volta dal nome vero e proprio della proprietà (e.g. `http://example.org/ont#prop`).

Vedremo poi come attraverso gli *strumenti* messi a disposizione da RDF Schema Language (RDFS) sarà possibile costruire delle ontologie anche molto *sofisticate* quali DAML+OIL. Sarà infatti con DAML+OIL che raggiungeremo lo strato dell'integrazione in cui sarà possibile far *parlare* tra loro ontologie diverse del Web semantico. In generale una ontologia può essere *costruita* sia partendo dalle sole classi e proprietà messe a disposizione da RDFS, sia facendo riferimento anche ad altre ontologie.

In realtà già in RDF esiste la possibilità di definire nuove proprietà, è infatti possibile asserire che una certa risorsa è istanza (`rdf:type`) della classe `rdf:Property`. Quello che non è possibile fare è dire quale sia il dominio ed il codominio di questa nuova proprietà o in che relazione sia con altre proprietà. Non è infatti possibile asserire

che la proprietà `mamma` è una *sottoproprietà* della più generale proprietà `genitore`, o che la proprietà `autore` ha come dominio i libri e codominio le persone.

Le ontologie che si andranno a costruire saranno scritte in RDF, saranno cioè fatte di asserzioni `(subject, predicate, object)`, ma sfruttando classi e proprietà base messe a disposizione da RDFS. In generale le ontologie *parlano* solo di classi, proprietà e delle loro gerarchie e non parlano degli *oggetti*. Sarà infatti in un *classico* documento RDF che, sfruttando le ontologie, si faranno asserzioni sulla realtà.

Le ontologie possono essere costruite per temi. Ad esempio sarà possibile costruire una ontologia che riguardi la tassonomia delle piante, un altra per la catalogazione dei pezzi di ricambio di un officina meccanica o più astrattamente si potranno creare delle ontologie per descrivere gli oggetti quali quelli della logica.

La differenza tra un DTD per XML ed un RDF Schema (i.e. ontologia) sta nel fatto che il primo serve per la validazione sintattica mentre il secondo fornisce significato alle asserzioni espresse in un modello RDF. Il cuore del vocabolario per la costruzione di Schemi RDF è associato normalmente al namespace `rdfs` identificato dalla URI `http://www.w3.org/2000/01/rdf-schema#`. A questo indirizzo si trova infatti un documento RDF che contiene il vocabolario del linguaggio RDF Schema Language. Infatti anche il vocabolario di base per costruire Schemi è definito attraverso asserzioni RDF. È per questo che all'interno di questo documento ci si riferisce anche al namespace `rdf` associato alla URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#` che a sua volta è uno Schema RDF per RDF stesso (ossia la definizione del suo vocabolario).

Tutte le definizioni ruotano attorno ai concetti di classe, risorsa e proprietà. In

RDFS una classe non viene descritta in termini delle proprietà che ha una sua istanza, ma le proprietà sono descritte in termini delle classi di risorse a cui quella proprietà può essere applicata. Questo è fatto mettendo dei vincoli sul `rdfs:range` e sul `rdfs:domain` a cui una certa proprietà può essere applicata. Per esempio in RDFS si definisce la proprietà `autore` dicendo che deve avere come `rdfs:domain` istanze della classe `Libro` e come `rdfs:range` i letterali. Ciò è diverso da quanto accade comunemente in un classico sistema Object Oriented in cui si definisce la classe `Libro` con un attributo chiamato `autore` di tipo letterale. Questo approccio basato sulle proprietà è più naturale sul Web semantico perché quando qualcuno vuole parlare di qualche risorsa che esiste ha la necessità di introdurre delle nuove proprietà per poterla descrivere. Quindi dirà che questa nuova proprietà serve per descrivere oggetti che siano istanze di una data classe. Questo meccanismo evita di dover ridefinire una classe ogni volta che si renda necessario aumentare la granularità della sua definizione.

1.8.1 Le Classi base di RDFS

Per *classe* intendiamo un *blueprint* ossia il *progetto* che definisce le caratteristiche comuni a tutti gli *oggetti* di un certo *tipo*. La classe non è l'insieme degli oggetti di un certo tipo né l'oggetto stesso, ne è solo la caratterizzazione, così come il progetto di una bicicletta non è né l'insieme delle biciclette né una bicicletta. Diremo che un oggetto è una *istanza* di una classe, non un suo membro. Una classe è una sorta di *stampo* per creare oggetti con certe caratteristiche. La classe `Cane` non è ne un *cane* ne l'insieme dei cani, mentre *Pippo* che è una istanza della classe `Cane` è un *cane*.

Nel vocabolario base di RDF Schema Language, al quale ci si riferisce con il

namespace `rdfs`, sono già definite delle classi base che servono per costruire delle altre classi. Le nuove classi saranno descritte solo in termini della loro gerarchia, non sono specificate al loro interno le loro proprietà. Saranno invece le definizioni delle proprietà che, dicendo che possono essere applicate solo a certe classi e che potranno assumere solo certi valori, definiranno implicitamente le *caratteristiche* di una classe.

Definizione 1.8.1. (URI qualificate)
Una URI che termini con il carattere `#` seguito da un `nome` è detta URI *qualificata*.

Commento 1.8.1. Le URI qualificate sono molto utili perché permettono con lo stesso prefisso di identificare più oggetti definiti nella stessa pagina. Tutte le classi base e le proprietà base di RDFS sono nominate attraverso delle URI qualificate e sono tutte definite all'interno dello stesso documento.

Le classi base di RDFS sono `rdfs:Class`, `rdfs:Resource` e `rdf:Property`. Osserviamo che in considerazione del fatto che:

```
rdfs = http://www.w3.org/2000/01/rdf-schema#
rdf  = http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

le classi predefinite:

```
rdfs:Class    = http://www.w3.org/2000/01/rdf-schema#Class
rdfs:Resource = http://www.w3.org/2000/01/rdf-schema#Resource
rdf:Property  = http://www.w3.org/1999/02/22-rdf-syntax-ns#Property
```

sono esse stesse delle risorse identificate da URI, sulle quali sarà quindi possibile fare delle asserzioni (e.g. quando vengono definite). Avremo che una istanza di `rdfs:Class` è una classe, una sottoclasse di `rdfs:Resource` è una classe di risorse (qualsiasi cosa nominabile con una URI) ed una istanza della classe `rdf:Property` sarà una proprietà.

Analogamente a quanto visto per le classi base avremo anche due risorse associate alle due principali proprietà predefinite.

`rdf:type        = http://www.w3.org/1999/02/22-rdf-syntax-ns#type`

`rdfs:subClassOf = http://www.w3.org/2000/01/rdf-schema#subClassOf`

In figura 1.8 a pagina 81 si può osservare come la classe `rdfs:Class` sia una classe definita come istanza di se stessa e di come sia la classe `rdfs:Class` che la classe `rdf:Property` siano sottoclassi della classe delle risorse. In questo modo classi che siano istanze della classe `rdfs:Class` e proprietà che siano istanze della classe `rdf:Property` saranno *automaticamente* delle risorse. Ciò sarà importante quando daremo la semantica esplicita per la traduzione in SMODELS di queste classi e proprietà.

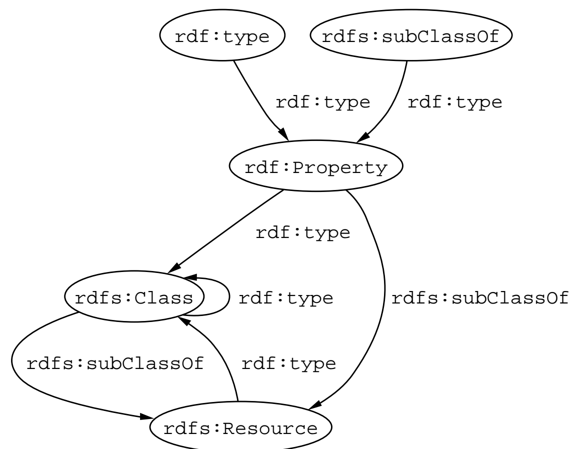


Figura 1.8: Grafo delle relazioni tra le classi e le proprietà base di RDFS.

La proprietà `rdf:type` può essere letta come *essere istanza di* che quindi risulta essere una sorta di `is_a` delle reti semantiche. La differenza rispetto alle reti semantiche che usavano solo la `is_a` per definire istanze di classi e definire sottoclassi di

classi sta nel fatto che per la definizione di sottoclassi qui si usa la proprietà speciale `rdfs:subClassOf`.

Per completezza e per riferimento in Appendice A è riportata la rete semantica di tutte le proprietà e classi di RDF ed RDFS relazionate tra loro in termini delle relazioni *essere istanza di* (`rdf:type`) ed *essere sottoclasse di* (`rdfs:subClassOf`).
Vediamo ora più formalmente e con degli esempi quanto detto.

Definizione 1.8.2. (Classe RDFS)
La risorsa identificata dalla URI `rdfs:Class` rappresenta una classe predefinita e serve per riferirsi all'usuale concetto di classe inteso come categoria. Negli Schemi RDF si possono definire delle nuove classi dichiarando che una certa risorsa `online`, definita tramite l'attributo `rdf:ID` del tag `rdf:Description`, è di tipo (`rdf:type`) `rdfs:Class`.

Esempio 1.8.1. Vediamo ora come istanziare una nuova classe. Pensiamo di essere interessati a creare una ontologia che *parli* di *veicoli a motore*. Per fare ciò definiamo la classe `Veicolo` nel modo seguente:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<rdf:Description rdf:ID="Veicolo">
 <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>
</rdf:RDF>
```

Di cui in figura 1.9 a pagina 83 la rappresentazione attraverso un grafo orientato.

Commento 1.8.2. Esiste un modo equivalente, ma molto più compatto e molto più usato, per istanziare classi e proprietà, ed è il seguente:

Figura 1.9: `Veicolo` è una classe [esempio: 1.8.1].

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdfs:Class rdf:ID="Veicolo"/>
</rdf:RDF>
```

In cui tutto è ridotto ad una sola linea in cui è implicitamente asserito che `Veicolo` è `rdf:type` della classe `rdfs:Class` come in figura 1.9 a pagina 83. In generale un tag del tipo:

```
<NomeClasse rdf:about="nome"/>
```

si legge come: la risorsa `nome` è istanza della classe `NomeClasse`.

Una nuova classe può essere definita per rappresentare qualsiasi cosa sul Web semantico (e.g. professori, concetti astratti o anche intere pagine Web). Una risorsa dichiarata di tipo `rdfs:Class` sarà a sua volta una classe (i.e. una istanza della classe `rdfs:Class` è un oggetto di tipo classe, quindi una classe).

Definizione 1.8.3. (Risorsa RDFS)
La classe `rdfs:Resource` è la classe base, nel senso che tutte le altre classi sono sottoclassi della classe identificata dalla URI `rdfs:Resource`.

Commento 1.8.3. Ciò è dovuto al fatto che tutto è una risorsa, nel senso che le asserzioni RDF si fanno solo su risorse. Un qualsiasi *oggetto*, sia esso un `subject` un `predicate` o un `object` nominato tramite una URI deve essere considerato come

istanza della classe `rdfs:Resource`. Il fatto di aver dichiarato che sia `rdfs:Class` che `rdf:Property` sono sottoclassi della classe `rdfs:Resource` farà sì che tutti gli oggetti istanza di nuove classi e di nuove sottoclassi di proprietà erediteranno il fatto di essere essi stesse delle risorse. Vedremo nel capitolo 4 come ciò verrà reso esplicito nella nostra semantica data per traduzione verso SMODELS.

Definizione 1.8.4. (Proprietà RDF)
La classe predefinita `rdf:Property` è la classe delle proprietà.

Commento 1.8.4. La classe `rdf:Property` serve per poter dichiarare che una certa risorsa (istanza della classe rdfs:Resource) è una proprietà o che è una sottoclasse di proprietà specializzate. Avremo quindi che istanze della classe `rdf:Property` sono delle proprietà, mentre sottoclassi saranno classi di proprietà. Visto che una proprietà è sempre una risorsa si intuisce il perché la classe `rdf:Property` sia una sottoclasse della classe `rdfs:Resource`. Osserviamo inoltre che il namespace di `rdf:Property` non è `rdfs` ma `rdf`.

Esempio 1.8.2. Se volessimo introdurre la nuova proprietà `guidare` potremmo farlo nel seguente modo:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<rdf:Description rdf:ID="guidare">
   <rdf:type
    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Property"/>
</rdf:Description>
</rdf:RDF>
```

1.8.2 Le Proprietà base di RDFS

Abbiamo detto che una proprietà è una istanza della classe `rdf:Property`. Esistono all'interno dei namespace `rdf` e `rdfs` delle proprietà predefinite che serviranno per costruire nuove classi e nuove proprietà o *raffinare* la definizione di classi e proprietà precedentemente definite.

Definizione 1.8.5. (`rdf:type`)
La proprietà `rdf:type` è una proprietà predefinita nel namespace `rdf` e serve per dire che una certa risorsa, istanza della classe `rdfs:Resource`, è una *istanza* di una certa classe. In una asserzione RDF che ha `rdf:type` come `predicate` deve essere che la risorsa che rappresenta l'`object` sia una classe, ossia una istanza della classe `rdfs:Class` (n.b. anche `rdfs:Class` è istanza di `rdfs:Class`).

Commento 1.8.5. Se diciamo che una certa risorsa è di `rdf:type` `rdfs:Class`, intendiamo dire che quella che abbiamo appena definito è una risorsa che rappresenta una classe e non un *oggetto*. Le istanze della classe `rdfs:Class` sono a loro volta delle classi.

Esempio 1.8.3. Se diciamo che la risorsa `Veicolo` è di tipo `rdfs:Class`, diciamo che `Veicolo` è l'identificativo di una nuova classe, ossia istanza di `rdfs:Class`. Mentre se diciamo che la risorsa `WVGolfTN616021` è di tipo `Veicolo`, stimo istanziando un *oggetto* della classe `Veicolo`.

Esempio 1.8.4. Quando si asserisce che una risorsa è *sottoclasse* di una classe già definita (e.g. `MiniVan` è sottoclasse di `Van`) si deve comunque dire che la risorsa che rappresenta la sottoclasse e.g. `MiniVan` sia di tipo `rdfs:Class`, questo per far sì che quando si istanzia un oggetto della sottoclasse (e.g. `minivanMario` è un `MiniVan`) la

proprietà `rdf:type` abbia come `object` una classe istanza della classe `rdfs:Class` (e.g. la classe `MiniVan`).

Definizione 1.8.6. (`rdfs:subClassOf`)
La proprietà `rdfs:subClassOf` è una proprietà predefinita che serve per dire che una certa risorsa, istanza della classe `rdfs:Class`, è una *sottoclasse* di una certa classe che sia anch'essa istanza della classe `rdfs:Class`. La proprietà `rdfs:subClassOf` gode della proprietà transitiva sia sulle classi che sulle loro istanze. Una classe può essere descritta come sottoclasse di più classi.

Commento 1.8.6. La transitività sulle istanze deve essere intesa nel seguente modo: se una risorsa `nome` è istanza della classe `A` che a sua volta è sottoclasse della classe `B` risulta che la risorsa `x` può essere vista come istanza della classe `B`, senza però esserlo direttamente. Il fatto che una classe può essere descritta come sottoclasse di più classi permette l'ereditarietà multipla.

Esempio 1.8.5. Potremmo asserire che la risorsa `Camion` (dichiarata essere di tipo `rdfs:Class`) è `rdfs:subClassOf` della risorsa `Veicolo` che a sua volta è una istanza della classe `rdfs:Class`.

Esempio 1.8.6. La classe `MiniVan` è sottoclasse sia della classe dei `VeicoliPasseggeri` che della classe `Van`, che a loro volta sono sottoclassi della classe `Veicoli`. Si può quindi concludere per la proprietà transitiva, di cui gode la proprietà `rdfs:subClassOf`, che anche la classe dei `MiniVan` è sottoclasse della classe dei `Veicoli`.

Definizione 1.8.7. (`rdfs:subPropertyOf`)
La proprietà `rdfs:subPropertyOf`, essendo una proprietà, è una istanza della classe `rdf:Property` e serve per asserire che una certa proprietà è la specializzazione di un

altra proprietà. La proprietà `rdfs:subPropertyOf` gode della proprietà transitiva. Una sottoproprietà eredita sia il dominio che il codominio della proprietà di cui è sottoproprietà. Non vi possono essere cicli nelle definizioni di sottoproprietà.

Commento 1.8.7. Il fatto che non siano ammessi cicli significa che non è possibile definire una sottoproprietà di una proprietà che sia a sua volta sottoproprietà della sottoproprietà che si sta definendo. Nel gergo delle relazioni una sottoproprietà è un sottoinsieme dell'insieme definito implicitamente dalla relazione di cui essa è sottoproprietà, quindi un sottoinsieme non può contenere l'insieme in cui esso è contenuto.

Esempio 1.8.7. La proprietà `mamma` è una *specializzazione* della proprietà `genitore` e ciò si esprime dicendo che ciò che è in relazione attraverso la proprietà `mamma` deve esserlo anche attraverso la proprietà `genitore`.

Definizione 1.8.8. (`rdfs:domain`)

La proprietà con nome `rdfs:domain` è una proprietà predefinita, istanza della classe `rdf:Property`, e serve per asserire che una risorsa che sia istanza della classe `rdf:Property` ha come *dominio* istanze di una certa classe. Una stessa proprietà può avere più di una classe come dominio, il *dominio* sarà la loro unione[5]. La proprietà `rdfs:domain` ha come `rdfs:domain` la classe `rdf:Property` e come `rdfs:range`[6] la classe `rdfs:Class`.

Commento 1.8.8. L'`object` di una asserzione che ha come `predicate` la proprietà `rdfs:domain` deve essere una istanza della classe `rdfs:Class`. In questo modo sarà

[5]Da informazioni recenti pare sia in corso una revisione di questa specifica da parte del W3C. La nuova specifica dovrebbe prevedere che il dominio sia definito come l'intersezione e non più come l'unione.

[6]Vedi oltre per la sua definizione.

possibile specificare che una certa proprietà può descrivere solo istanze di una certa classe.

Esempio 1.8.8. La proprietà `autore` può avere come dominio sia istanze della classe `Canzone` che istanze della classe `Libro`.

Definizione 1.8.9. (`rdfs:range`)
La proprietà con nome `rdfs:range` è una proprietà predefinita, istanza della classe `rdf:Property`, e serve per asserire che una risorsa che sia istanza della classe `rdf:Property` ha come *codominio* istanze di una certa classe istanza della classe `rdfs:Class`. Una stessa proprietà può avere più di una classe come codominio, il *codominio* sarà la loro unione[7]. La proprietà `rdfs:range` ha come `rdfs:domain` la classe `rdf:Property` e come `rdfs:range` la classe `rdfs:Class`.

Commento 1.8.9. L'`object` di una asserzione che ha come `predicate` la proprietà `rdfs:range` deve essere una istanza della classe `rdfs:Class`. In questo modo sarà possibile specificare che una certa proprietà assume come valori solo istanze di una certa classe.

Esempio 1.8.9. La proprietà `autore` può avere come codominio sia istanze della classe `Uomini` che istanze della classe `Donne`.

Vi sono altre classi ed altre proprietà definite in RDF e RDFS che fin qui non abbiamo visto e che rivestono un ruolo marginale all'interno di questa trattazione. Vediamole qui brevemente:

- `rdfs:comment` è una proprietà che si utilizza per dare delle descrizioni verbali attraverso un letterale;

[7]Da informazioni recenti pare sia in corso una revisione di questa specifica da parte del W3C. La nuova specifica dovrebbe prevedere che il codominio sia definito come l'intersezione e non più come l'unione.

- `rdfs:label` è una proprietà che si utilizza per dare, tramite un letterale, una versione leggibile da un essere umano del nome della risorsa;
- `rdfs:seeAlso` è una proprietà che serve per fornire una risorsa alternativa che descrive la risorsa che si sta descrivendo;
- `rdfs:isDefinedBy` è una proprietà che serve per specificare il namespace all'interno del quale è definita la risorsa che si sta descrivendo;
- `rdf:value` è una proprietà e serve per identificare il principale valore (normalmente una stringa) di una proprietà quando i valori della proprietà sono strutturati;
- `rdfs:Literal` è una classe le cui istanze rappresentano gli elementi atomici, normalmente delle stringhe di caratteri.

Per completezza e per riferimento, in Appendice B è riportata la rete semantica di tutte le proprietà[8] di RDF ed RDFS relazionate tra loro in termini di relazioni *dominio* (`rdfs:domain`) e *codominio* (`rdfs:range`), mentre in Appendice C si può trovare la lista di tutte le classi e le proprietà definite in RDF ed RDFS in ordine di importanza.

1.8.3 Esempio di Schema RDFS

Vediamo ora un esempio che metta in pratica le nozioni fin qui acquisite relativamente ad RDF ed RDF Schema Language, con riferimento alla possibilità di creare delle ontologie (i.e. Schemi).

[8] La proprietà `rdf:object` non ha specificato nessun `rdfs:range` perché le specifiche del W3C prevedono che questa proprietà non abbia vincoli sul codominio.

Esempio 1.8.10. Interessati alla creazione di una ontologia che ci permetta di descrivere il *mondo* dei *veicoli a motore*, andiamo a definire una gerarchia di classi. Prima di tutto definiremo la classe `Veicolo`. Per fare ciò diciamo semplicemente che è di tipo `rdfs:Class` ossia è una classe. Facciamo anche una asserzione per dire che la classe `Veicolo` è una sottoclasse della classe delle risorse (`rdfs:Resourse`), questo perché quando diremo che un certa risorsa è istanza della classe `Veicolo` (per la transitività della proprietà `rdfs:subClassOf`) la nuova risorsa erediterà il fatto di essere proprio una risorsa. In questo modo l'istanza della classe `Veicolo`, essendo effettivamente una risorsa, potrà essere usata in una asserzione RDF. Le sottoclassi della classe `Veicolo` le dichiariamo essere delle classi esplicitamente, e non ci limitiamo a dire semplicemente che sono sottoclassi perché la proprietà `rdf:type` ha come `rdfs:range` istanze della classe `rdfs:Class`, e quindi per poter dire che un oggetto è una istanza ad esempio della classe `MiniVan` dovremo asserire esplicitamente che `MiniVan` sia una istanza della classe `rdfs:Class`. Asserendo che una nuova classe è sottoclasse della classe `Veicolo`, essendo questa sottoclasse della classe `rdfs:Resource`, per la transitività di `rdfs:subClassOf` le nuove classi saranno sottoclassi della classe `rdfs:Resource` e quindi le loro istanze saranno risorse. In questo modo potremo *usare* le istanze delle sottoclassi nelle asserzioni RDF. Il fatto che una sottoclasse sia essa stessa una risorsa è ereditato dal fatto che una sottoclasse è una istanza della classe `rdfs:Class` che a sua volta è sottoclasse della classe `rdfs:Resource`. Vediamo ora il documento RDF XML che codifica quanto da noi fin qui asserito verbalmente.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<!-- Questo Schema RDF si potrà utilizzare in un documento RDF
```

```
     riferendosi ad esso tramite una dichiarazione di namespace
     (e.g. xmlns:ont="http://example.org#"). Ciò permetterà di
     utilizzare delle abbreviazioni (ad esempio ont:Veicolo) per
     riferirci in modo non ambiguo alla classe dei veicoli a motore.
-->

<rdf:Description rdf:ID="Veicolo">
 <rdf:type
  rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf
   rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Description>

<rdf:Description rdf:ID="VeicoloPasseggeri">
 <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#Veicolo"/>
</rdf:Description>

<rdf:Description rdf:ID="Camion">
 <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#Veicolo"/>
</rdf:Description>

<rdf:Description rdf:ID="Van">
 <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#Veicolo"/>
</rdf:Description>

<rdf:Description rdf:ID="MiniVan">
 <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#Van"/>
 <rdfs:subClassOf rdf:resource="#VeicoloPasseggeri"/>
</rdf:Description>
</rdf:RDF>
```

Si può osservare in figura 1.10 a pagina 92 di come la classe `MiniVan` sia sottoclasse

di due classi diverse (i.e. c'è la possibilità di realizzare ereditarietà multipla).

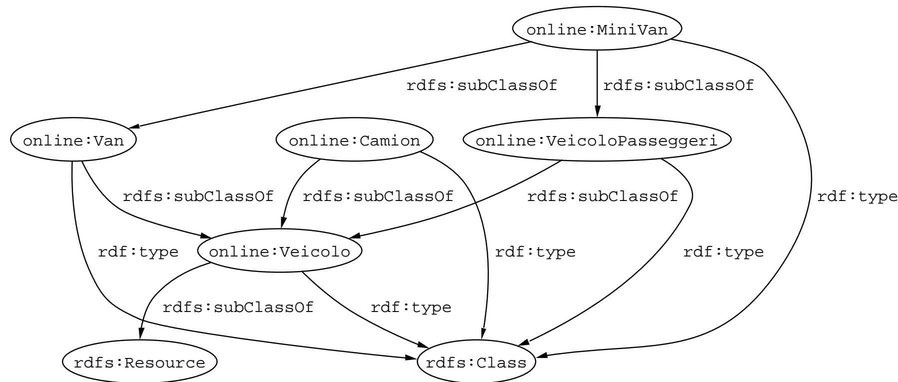


Figura 1.10: Classi e sottoclassi nel dominio semantico dei *veicoli* [esempio: 1.8.10].

A questo punto data questa ontologia, *domiciliata* sul Web semantico ad una data URL, sarà possibile riferirsi ad essa all'interno di un documento RDF XML per *istanziare* oggetti di queste classi.

Esempio 1.8.11. Pensando che l'ontologia appena descritta sia reperibile alla URL `http://example.org/Veicoli` asseriamo che la mia WVGolf è un *veicolo*:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<rdf:Description rdf:ID="LaGolfDiFranco">
  <rdf:type rdf:resource="http://example.org/Veicoli#Veicolo"/>
</rdf:Description>
```

Possiamo fare la stessa asserzione RDF in modo più compatto (quello più usato):

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ont="http://example.org/Veicoli#">
<ont:Veicolo rdf:ID="LaGolfDiFranco"/>
</rdf:RDF>
```

Dove abbiamo dichiarato un namespace e lo abbiamo poi utilizzato per *qualificare* la proprietà `Veicolo` che è la classe di cui la risorsa `online:LaGolfDiFranco` ne è istanza.

1.8.4 Gli Schemi di RDF ed RDFS

Come abbiamo già detto RDF è definito da una grammatica EBNF, ma le sue classi e le sue proprietà sono definite all'interno di una ontologia che fa uso delle classi e delle proprietà di RDFS. Questa ontologia è scritta in RDF XML ed è riportata in Appendice D.

Tutte le proprietà e le classi di RDFS sono a loro volta definite all'interno di una ontologia che fa uso delle classi e delle proprietà definite in RDFS ma anche in RDF. Questa ontologia è anch'essa scritta in RDF XML ed è riportata in Appendice E. Sulle classi e le proprietà descritte all'interno dello Schema, vengono fatte altre asserzioni per raffinarne le definizioni. Come esempio riportiamo qui la definizione completa di `rdfs:Class`:

```
<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Class">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">Class</rdfs:label>
  <rdfs:comment>The concept of Class</rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdfs:Class>
```

C'è da osservare che lo Schema di RDF e quello di RDFS sono fortemente legati l'uno all'altro e costituiscono, visti nella loro globalità, il vocabolario di base per il Web semantico. Ogni altra ontologia verrà costruita sfruttando i meccanismi di asserzioni RDF e le *classi* e le *proprietà* definite in `rdf` e `rdfs`.

Capitolo 2

DAML+OIL

Questo capitolo è dedicato a DAML+OIL e a motivare la nostra scelta di tale linguaggio, anche in vista delle estensioni a DAML for Services (DAML-S) e DAML Logic (DAML-L). Spiegheremo perché DAML+OIL è una ontologia e perché riveste in prospettiva un ruolo così importante per il Web semantico.

2.1 Introduzione a DAML+OIL

Nel Capitolo 1 abbiamo presentato tutta la struttura del Web semantico arrivando fino allo strato delle ontologie. Abbiamo infatti mostrato attraverso degli esempi di come sia possibile *costruire* delle ontologie (i.e. vocabolari per il Web semantico). Nonostante il fatto che RDFS sia sufficientemente generale per creare ontologie, è altresì vero che il numero ridotto di proprietà e classi predefinite lo rendono poco utilizzabile praticamente.

È doveroso infatti spendere due parole sull'*arte* della creazione di ontologie. Sebbene sia praticamente semplice asserire che una risorsa sia una classe e che un'altra risorsa sia una sottoclasse o una proprietà è altresì vero che la creazione di ontologie è

un'attività concettualmente complessa e raffinata. Usiamo come metafora quella dell'architettura e pensiamo a come sia difficile passare dalla conoscenza tecnica sull'uso del tecnigrafo alla progettazione del Museo Guggenheim di Bilbao.

La creazione di ontologie è una attività difficile che prevede sia profonde conoscenze sul *dominio* che si sta *modellizzando*, sia sugli strumenti linguistici che si stanno usando per farlo.

Se volessimo classificare le ontologie in due grandi famiglie, potremmo farlo individuando una prima famiglia di ontologie che potremmo chiamare di *dominio stretto* ed una seconda famiglia che chiameremo *ontologie generali*. Alla prima famiglia appartengono ontologie quali quella vista nell'esempio sui veicoli a motore (pagina 90) in cui ci si preoccupa di creare una tassonomia sì generale, ma per uno specifico dominio (i.e. quello dei veicoli a motore). Alla seconda famiglia appartengono ontologie che si preoccupano di fornire un vocabolario di uso così generale che praticamente diventi utilizzabile per creare in modo molto più agevole ed efficace le *ontologie di dominio*. A questa famiglia sicuramente appartiene RDFS ed RDF, ma come già accennato il loro vocabolario è troppo ristretto per essere praticamente usabile.

In questa tesi siamo interessati ad *ontologie generali*, ed è per questo che introduciamo l'ontologia DAML+OIL. Questa ontologia nasce con lo scopo di estendere RDFS fornendo nuovi strumenti linguistici per la descrizione del mondo e raffinando quelli già presenti in RDFS.

> "*The DARPA Agent Markup Language (DAML) Program officially began in August 2000. The goal of the DAML effort is to develop a language and tools to facilitate the concept of the Semantic Web*". – Murray Burke[1]

[1] Program Manager del DARPA

L'ontologia DAML+OIL nasce dalla fusione di DAML (DARPA Agent Markup Language), sviluppato in America per conto del DARPA (Defense Advanced Research Projects Agency), e di OIL (Ontology Inference Layer), sviluppato in Europa principalmente da un gruppo di lavoro tedesco.

Il nostro interesse nei confronti delle ontologie generali, ed in particolare per DAML+OIL, è dovuto al fatto che il nostro scopo è quello di esplicitare una semantica di base attraverso l'utilizzo della programmazione logica. Ricordiamo infatti che le ontologie sono degli strumenti fondamentalmente linguistici e allo stato attuale mancano di una semantica *forte* e standardizzata. È infatti oggi difficile dire se un documento marcato con una certa ontologia sia semanticamente corretto, e non è chiaro quali siano i fatti inferibili da esso. Vedremo nel Capito 4 la soluzione da noi proposta. Il nostro lavoro non andrà solo nella direzione di esplicitare una semantica per traduzione, ma cercherà di estenderla al fine di catturare possibili meccanismi inferenziali quali quelli del *ragionamento per default*.

2.2 Perché DAML+OIL?

Riprendiamo brevemente un esempio già fatto in precedenza ma che a questo punto risulterà sicuramente più chiaro per mostrare perché l'utilizzo di DAML+OIL risulterà vantaggioso rispetto a quello di XML.

DAML+OIL risulta essere più efficace rispetto ad XML perché dato un insieme di asserzioni DAML+OIL è possibile ricavare nuova conoscenza espressa tramite delle altre asserzioni DAML+OIL. Il tutto direttamente dalle asserzioni originali espresse in DAML+OIL e non, come succederebbe in XML, attraverso *conoscenze* contenute in uno specifico programma esterno.

Con i modi usuali di RDF pensiamo di asserire che `mamma` è una *sottoproprietà* della proprietà `genitore` e di asserire che *Pia è mamma di Franco.* Un sistema inferenziale basato DAML+OIL, a fronte della domanda *Chi sono i genitori di Franco?* potrà rispondere *Pia è genitore di Franco* e questo senza che ciò sia stato asserito esplicitamente. Ciò non sarebbe possibile in XML in quanto in esso non è definita nessuna semantica. Per farlo si potrebbe costruire un programma C++ che avesse in sé la semantica voluta, ma questa sarebbe specifica per quella particolare situazione operativa e non sarebbe generalizzabile come invece sarà possibile per DAML+OIL.

La situazione appena descritta potrebbe essere catturata direttamente tramite RDF Schema Language costruendo uno Schema in cui attraverso una asserzione RDF si esprime il fatto che `mamma` è una `rdfs:subPropertyOf` della proprietà `genitore`; con DAML+OIL si potranno fare delle deduzioni più sofisticate, visto che sono predefinite delle proprietà più raffinate e specializzate di quelle messe a disposizione da RDFS. Ad esempio in DAML+OIL sono predefinite delle proprietà di *equivalenza*, nonché la classe delle proprietà che possono assumere un solo valore (e.g. il numero di passaporto).

2.3 L'ontologia DAML+OIL

DAML+OIL essendo una ontologia basata su RDFS è definito tramite uno Schema RDFS, di cui in Appendice F è riportata l'ultima versione rilasciata. Essendo una ontologia alla quale ci si può riferire per costruire altre ontologie questa è reperibile sul Web alla URL:

`http://www.daml.org/2001/03/daml+oil#`

alla quale ci riferiremo attraverso il namespace `daml`, analogamente a quanto fatto per `rdf` ed `rdfs`. Nel *leggere* l'ontologia DAML+OIL non c'è da stupirsi che all'inizio di essa vengano dichiarati, per poi essere usati, i due namespace `rdf` ed `rdfs`. Ciò è dovuto al fatto che una ontologia è scritta attraverso asserzioni RDF, per cui si necessitano dei nomi definiti nel namespace `rdf`, e si usano classi e proprietà base definite da RDFS nello Schema `rdfs`.

Vediamo ora attraverso un semplice esempio un primo uso di una delle *nuove* classi definire da DAML+OIL.

Esempio 2.3.1. (`daml:DatatypeProperty`)

Vogliamo definire una proprietà `peso` che abbia come dominio la classe `Persona` e per codominio la classe `NumeroIntero`. Usando direttamente RDFS si potrebbe fare nel seguente modo:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdfs:Class rdf:ID="Persona"/>
<rdfs:Class rdf:ID="NumeroIntero"/>
<rdf:Property rdf:ID="peso">
    <rdfs:domain rdf:resource="#Persona"/>
    <rdfs:range rdf:resource="#NumeroIntero"/>
</rdf:Property>
</rdf:RDF>
```

Per fare ciò abbiamo dovuto definire la classe `NumeroIntero` per poter asserire che la proprietà `peso` ha come codominio quello dei *numeri interi* e non un generico letterale. In considerazione del fatto che capita spesso di avere delle proprietà che associano un valore `numerico` ad una risorsa quello che è stato fatto in DAML+OIL è stato di creare la classe specializzata `daml:DatatypeProperty`, definita come sottoclasse di

`rdf:Property`, che serve per definire proprietà che associano dati tipizzati a risorse. Ciò è stato fatto all'interno del namespace `daml` nel seguente modo:

```
<rdfs:Class rdf:ID="DatatypeProperty">
  <rdfs:label>DatatypeProperty</rdfs:label>
  <rdfs:comment>
    if P is a DatatypeProperty, and P(x, y), then y is a data value.
  </rdfs:comment>
  <rdfs:subClassOf
    rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>
```

DAML+OIL permette quindi di definire delle proprietà che abbiano come codominio classi che rappresentano dati tipizzati definiti. Le classi che dovranno essere usate come codominio di proprietà che siano istanze della classe `daml:DataTypeProperty` sono definite dall'utente o sono presenti nell'ontologia XSDL.

Ritornando all'esempio precedente, riscriviamo la proprietà `peso` come istanza della classe `daml:DatatypeProperty` con `rdfs:range` la classe dei numeri interi predefinita in XSDL.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
<rdfs:Class rdf:ID="Persona"/>
<daml:DatatypeProperty rdf:ID="peso">
 <rdfs:domain rdf:resource="#Persona"/>
 <rdfs:range
   rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</daml:DatatypeProperty>
</rdf:RDF>
```

La classe `http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger` non appartiene al *mondo* RDF bensì a quello XML. A questo livello di descrizione però non

ci interessiamo di come siano definiti gli schemi XML, sfruttiamo solo il fatto che al loro interno è stato definito un `simpleType` di nome `nonNegativeInteger` che noi utilizzeremo come se fosse una classe per descrivere i numeri interi non negativi.

Commento 2.3.1. (`daml:ObjectProperty`)
Sappiamo però che non tutte le proprietà hanno codomini *tipizzati*, tante proprietà hanno per codominio classi generiche. È per questo che in DAML+OIL è stata introdotta la classe `daml:ObjectProperty` che è sottoclasse della classe `rdf:Property` e che serve per definire proprietà che assumono come valori *oggetti* generici.

Abbiamo detto nel Capitolo 1 che non vi è modo di *capire* se due nomi diversi identificano lo stesso oggetto. Può capitare che in due ontologie diverse vengano definite due classi diverse che però identificano praticamente gli stessi oggetti. Questo problema può essere parzialmente risolto grazie a DAML+OIL. È infatti possibile, una volta capito che due nomi indicano la stessa cosa, asserire che i due nomi in realtà rappresentano lo stesso oggetto. Questa capacità espressiva di DAML+OIL permette all'atto pratico di integrare tra loro le ontologie senza troppa fatica.

Definizione 2.3.1. (`daml:samePropertyAs`)
La proprietà `daml:samePropertyAs` è definita come istanza della classe `rdf:Property` e come sottoproprietà della proprietà `rdfs:subPropertyOf` e serve per asserire che due nomi di proprietà indicano la stessa proprietà.

Commento 2.3.2. La proprietà `daml:samePropertyAs` risulta particolarmente utile dopo che sia stata rilasciata una ontologia DAML+OIL. Sarà infatti possibile senza modificare le pagine marcate con quella ontologia asserire, all'interno dell'ontologia

stessa, che una proprietà definita in un altra ontologia è la stessa proprietà di una presente nell'ontologia.

Esempio 2.3.2. Supponiamo di avere nel namespace `http://example.org/ont` la proprietà `weight` e di voler asserire che la proprietà `peso` della nostra ontologia è equivalente alla proprietà `weight` in modo da *armonizzare* le due ontologie. Ciò può essere fatto con un minimo sforzo implementativo semplicemente aggiungendo l'asserzione seguente alla nostra ontologia.

```
<rdf:Description rdf:about="#Peso">
  <daml:samePropertyAs rdf:resource="http://example.org/ont#weight"/>
</rdf:Description>
```

Una situazione operativa comune è quella di descrivere una risorsa attraverso una proprietà che può assumere un solo valore all'interno del suo codominio (e.g. codice fiscale). DAML+OIL risolve questo problema definendo la classe delle proprietà che possono assumere un solo valore.

Definizione 2.3.2. (`daml:UniqueProperty`)
La classe `daml:UniqueProperty` è definita come sottoclasse della classe `rdf:Property` e serve per istanziare proprietà che su una data risorsa assumano un solo valore all'interno del loro codominio.

Commento 2.3.3. Se `p` è una proprietà istanza della classe `daml:UniqueProperty` e vi sono due asserzioni del tipo `<x> <p> <y>` e `<x> <p> <z>` allora `y=z`.

Esempio 2.3.3. Il numero di scarpe di una persona è unico per ogni persona. Per asserire ciò basterà definire la proprietà `numeroScarpe` dicendo che è istanza sia della classe `daml:UniqueProperty` sia della classe `daml:DatatypeProperty` (esempio di

ereditarietà multipla). La proprietà `numeroScarpe` avrà sia la caratteristica di essere unica che di assumere solo valori tipizzati.

```
<daml:DatatypeProperty rdf:about="numeroScarpe">
   <rdf:type
    rdf:resource="http://www.w3.org/2001/10/daml+oil#UniqueProperty"/>
</daml:DatatypeProperty>
```

È importante osservare che l'aver detto che la proprietà `numeroScarpe` è *unica* non significa che si può fare una sola asserzione, ma solo che l'`object` per un dato `subject` deve essere unico.

Non è situazione rara quella di definire proprietà che sono transitive, la stessa `rdfs:subClassOf` è una proprietà transitiva. Per fornire uno strumento che descriva questa situazione DAML+OIL mette a disposizione la classe `daml:Transitive`

Una forma primitiva di negazione è fornita dalla proprietà `daml:complementOf` che permette di asserire che due classi sono l'una il complemento dell'altra. Una proprietà più generale della `daml:complementOf` è la proprietà `daml:disjointWith`, che permettere di asserire che due classi sono tra loro disgiunte. Esplicitare la semantica di questa proprietà permetterà di concludere che Socrate non è una donna visto che Socrate è un uomo e la classe delle donne è disgiunta da quella degli uomini.

2.4 DAML-S

Come esempio dell'utilità di DAML+OIL citiamo l'ontologia DAML-S (DAML for Services). Abbiamo parlato nel primo capitolo dei servizi e della necessità da parte dell'agente del *cliente* e di quello del *fornitore di servizio* di poter *comunicare* tra loro

per raggiungere uno scopo comune. L'idea è quella che i servizi (come ogni altra cosa sul Web semantico) siano descritti attraverso l'uso di ontologie. DAML-S riveste un ruolo particolarmente interessante perché definisce l'ontologia base per la definizione di tutti i servizi.

I servizi hanno tra loro delle caratteristiche comuni che sono catturate appunto dall'ontologia DAML-S. Ecco che, in considerazione che un servizio avviene almeno tra due entità, esiste la classe `damls:Actor`, le cui istanze sono il ricevente o il fornitore del servizio. Il fatto che un servizio sarà disponibile solo all'interno di una certa area, è descritto tramite la classe `damls:location`. Visto che un servizio sarà caratterizzato da dei parametri (e.g. accetta carte di credito) esiste la proprietà `profile:serviceParameter`.

2.5 DAML-L

Lo strato successivo a quello delle conversioni (DAML+OIL) è quello logico. Vi sono vari lavori in atto in questo settore (e.g. RuleML), ma quello più significativo dovrebbe essere DAML-L (DAML for Logic).

Questa ontologia non è stata ancora rilasciata, ma vi è tutta una serie di problemi ai quali DAML-L vorrebbe dare risposta.

- la negazione sicuramente è uno `strumento` utile e potente per descrivere la realtà, e quindi utile per il Web semantico, ma l'usuale negazione da fallimento non può essere implementata perché non sussiste la possibilità di esplorare tutto il Web per dire che una certa cosa non essendo stata asserita è da considerarsi

falsa. Le possibile soluzioni proposte sono una negazione da fallimento contestuale ed una per time-out. Oltre alla negazione da fallimento sarebbe utile definire la semantica della negazione totale (i.e. esplicita);

- i defaults sono molto spesso modi veloci per rappresentare la conoscenza sul mondo (e.g. gli uccelli normalmente volano) evitando così di elencare tutte le classi che godono di una certa proprietà (li vedremo nel Capitolo 6).

Capitolo 3

Programmazione logica: Answer Set Programming

In questo capitolo viene presentato l'Answer Set Programming [5] (ASP), le sue definizioni, ed il risolutore SMODELS [9] usato per la parte implementativa della tesi.

3.1 Modelli stabili

3.1.1 Sintassi

Consideriamo un linguaggio di costanti e predicati costanti. Assumiamo inoltre che termini e atomi siano costruiti come nel corrispondente linguaggio del primo ordine.

In quanto segue, consideriamo esclusivamente teorie *pseudoproposizionali* dove, cioè, possiamo sostituire iterativamente le variabili con i simboli di costante. Inoltre, diversamente dalla logica classica e dalla programmazione logica standard, non sono consentiti simboli di funzioni.

Una *regola* è definita come un'espressione della forma:

$$\rho : A_0 \leftarrow A_1, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n \qquad (3.1.1)$$

dove $A_1, \ldots, A_m$ sono atomi e *not* è un connettivo logico chiamato *negazione da fallimento*.

Per ogni regola definiamo

- $head(\rho) = A_0$,
- $pos(\rho) = A_1, \ldots, A_m$,
- $neg(\rho) = A_{m+1}, \ldots, A_n$
- $body(\rho) = pos(\rho) \cup neg(\rho)$.

La testa delle regole non è mai vuota, se $body(\rho) = \emptyset$ ci riferiamo a ρ come ad un *fatto*.

Un programma logico è definito come una collezione di regole. Una regola con variabili è considerata come abbreviazione dell'insieme di tutte le sue istanziazioni; l'insieme di tutti gli atomi *ground* che possiamo formare con il linguaggio associato al programma Π sarà denotato con B_Π (la base di Herbrand di Π).

Le *query* ed i *vincoli* sono espressioni con la stessa struttura delle regole ma con la testa vuota.

3.1.2 Semantica

Intuitivamente, un modello stabile è una *"visione del mondo"* compatibile con le regole del programma; le regole sono cioè viste come vincoli su questa visione del mondo.

Iniziamo a definire i modelli stabili della sottoclasse dei programmi positivi, ovvero quelli per cui, per ogni regola ρ, $neg(\rho) = \emptyset$.

Definizione 3.1.1. (Modelli stabili di programmi positivi)
Il modello stabile $a(\Pi)$ di un programma positivo Π è il più piccolo sottoinsieme di B_Π tale che per ogni regola 3.1.1 in Π:

$$A_1, \ldots, A_m \in a(\Pi) \Rightarrow A_0 \in a(\Pi) \tag{3.1.2}$$

Chiaramente i programmi positivi hanno un unico modello stabile che coincide con quello che si ottiene applicando altre semantiche; in altre parole i programmi positivi sono non ambigui. Il modello stabile è correlato al modello minimale della teoria classica che si può associare a Π.

Definizione 3.1.2. (Modelli stabili di programmi)
Sia Π un programma logico. Per ogni insieme S di atomi , sia $\Gamma(\Pi, S)$ un programma ottenuto da Π eliminando

(I) ogni regola che ha una formula 'not A' nel suo corpo, con $A \in S$;

(II) tutte le clausole della forma 'not A' nel corpo delle rimanenti regole

Chiaramente $\Gamma(\Pi, S)$ non contiene 'not', così il suo modello stabile è già definito. Se questo modello stabile coincide con S, allora diciamo che S è un modello stabile di Π. In altre parole, un modello stabile di Π è caratterizzato dall'equazione:

$$S = a(\Gamma(\Pi, S)). \tag{3.1.3}$$

I programmi che hanno un solo modello stabile vengono chiamati *categorici*.
Possiamo ora definire l'implicazione logica ($\models$) nella semantica dei modelli stabili.

Un atomo *ground* α è *vero* in S se $\alpha \in S$, *falso* altrimenti, ovvero $\neg\,\alpha$ è vero in S. Diciamo che ϕ *è un teorema di* Π (scritto $\Pi \models \phi$) se ϕ è vera in tutti i modelli stabili di Π.

Diremo che la risposta alla query γ è

si se γ è vera in tutti i modelli stabili di Π, cioè $\Pi \models \gamma$

no se $\neg\gamma$ è vera in tutti i modelli stabili di Π, cioè $\Pi \models \neg\gamma$

indefinita altrimenti

È facile osservare che i programmi logici sono non-monotoni nel senso che, aggiungendo nuove informazioni al programma, potremmo dover ritrattare qualche conclusione precedente.

3.1.3 Answer Set Programming

La semantica vista nella sezione precedente è alla base della *"programmazione per insiemi di risposte"* (Answer Set Programming o ASP). Infatti, la Definizione 3.1.1 dà un significato dichiarativo alla negazione da fallimento e stabilisce una connessione diretta con la logica di Reiter e con altri rilevanti formalismi di ragionamento non-monotono. La semantica AS tratta correttamente le definizioni cicliche negative selezionando alcuni dei modelli classici della teoria (in particolare quelli che sono *stabili*). Viceversa, nella semantica "Well-Founded" [4] tutti gli atomi coinvolti in cicli negativi sono considerati *indefiniti.*

Bisogna osservare che in ASP le soluzioni per un problema sono rappresentate da "insiemi di risposte" (*answer sets*) e non da istanziazioni di variabili prodotte in risposta ad una query, come invece nella programmazione logica tradizionale.

3.2 Alcuni esempi

Esempio 3.2.1. Consideriamo il problema di assegnare 3 colori (rosso, blu e verde) ai vertici di un grafo in modo che vertici adiacenti non abbiano lo stesso colore.
Questo problema, noto in letteratura come 3COL è *NP-Completo*, cioè intrattabile per input di grandi dimensioni.

Vediamo come dovrebbero configurarsi l'input e l'output di un programma ASP per la 3-colorazione di grafi.
Per prima cosa, vediamo la rappresentazione del grafo con fatti:

```
node(0..3).

col(red).
col(blue).
col(green).

edge(0,1).
edge(1,2).
edge(2,0).
edge(1,3).
edge(2,3).
```

... Motore d'inferenza ...

Supponiamo ora di chiamare un interprete per ASP, per esempio, SMODELS. L'output desiderato è del tipo:

```
{color(0,red), color(1,blue), color(green), color(3,red)}
{color(0,red), color(1,green), color(blue), color(3,red)}
{color(0,blue), color(1,red), color(green), color(3,blue)}
{color(0,blue), color(1,green), color(red), color(3,blue)}
{color(0,green), color(1,blue), color(red), color(3,green)}
{color(0,green), color(1,red), color(blue), color(3,green)}
```

Come costruire un programma logico per cui il calcolo, sia esso costruzione di modelli stabili in ASP o sia esso risposta a query come in Prolog, restituisca esattamente le 3-colorazioni descritte sopra?

Vediamo due programmi esempio che illustrano la differenza fra programmazione in ASP e programmazione in Prolog.

Esempio 3.2.2. (3-colorazione in Prolog)
Questo esempio di programma prolog per la 3COL serve a mostrare le differenze fra i due stili.
In Prolog, il calcolo corrisponde a delineare una lista di associazioni nodo/colore.
In ASP ogni modello stabile corrisponde ad un assegnamento che rispetta i vincoli dell'istanza.

```
graph_coloring(G,L):- gr_col(G,[],L).

gr_col([],L,L):-!.
gr_col([N|R],P,L):- vertex_color(N,C,P),
                    gr_col(R, [color(N,C)|P],L).

vertex_color(N,C,P):- col(C), ok_color(N,C,P).

neighbourhood(N,B,L):- (edge(N,N1);edge(N1,N)),
                       member(N1,B), !,
                       neighbourhood(N,[N1|B],L).
neighbourhood(_,L,L).

ok_color(N,C,P):- neighbourhood(N,[],L),
                  check_color(L,C,P).

check_color([],_,_):-!.
check_color([N1|B],C,P):- \ + member(color(N1,C),P),
                          !,
```

```
                              check_color(B,C,P).

member(E, [E|X]):- !.
member(E, [_|X]):- member(E,X).
```

Vediamo ora una sessione tipica (*):

```
?- consult[3col.pro]
?- graph_coloring([0,1,2,3],L).
L = [color(0,red), color(1,blue), color(green), color(3,red)];
L = [color(0,red), color(1,green), color(blue), color(3,red)];
...
```

Esempio 3.2.3. (3-colorazione in ASP - continuazione dell'esempio 3.2.1)

Vediamo come descrivere il problema della 3-colorazione in ASP.

```
color(X,red) | color(X,blue) | color(X,green) :- node(X).

:- edge(X,Y), col(C), color(X,C), color(Y,C).

hide node(X).
hide edge(X,Y).
hide col(C).

(*) Con il comando:
      lparse < 3col.txt | smodels 0
otteniamo:

Answer1
{color(0,red), color(1,blue), color(green), color(3,red)}
Answer2
{color(0,red), color(1,green), color(blue), color(3,red)}
...
```

3.3 Implementazioni esistenti

Il calcolo dei modelli stabili è oggi un'area di fervida ricerca. Sono disponibili implementazioni piuttosto efficienti e competitive coi migliori risolutori per SAT, che vengono impiegati anche industrialmente nella verifica di sistemi (model checking).

Queste implementazioni risolvono la versione *esatta* del problema e sono orientate al calcolo di tutti i modelli stabili del programma dato, cioè all'esplorazione dell'intero spazio (albero di decisione). D'altra parte, il nostro sistema è dedicato alla verifica dell'esistenza di *un* modello, senza esplorazione esaustiva. Comunque, nel caso del model checking, questa caratteristica non viene percepita come una limitazione.

Allo stato attuale, le implementazioni più competitive in termini di performance sono SMODELS (sviluppato dalla Helsinki University of Technologies) e DLV (sviluppato da TU Wien e Università della Calabria). Per i dettagli di queste e altre implementazioni si faccia riferimento a [11]. Mentre SMODELS è un risolutore per ASP che accetta la sintassi dei programmi logici standard, DLV tratta il sovrainsieme detto dei *Programmi logici disgiuntivi* definito da Gelfond e Lifschitz [6].

Nel frammento proposizionale, cioè quello senza i simboli di funzione, i programmi logici disgiuntivi sono strettamente più espressivi di quelli standard. Infatti, la complessità del problema DASP (ASP per programmi disgiuntivi) è maggiore di NP: DASP è un problema completo per il terzo livello della gerarchia polinomiale. Sperimentalmente, su programmi logici standard DLV è l'implementazione più efficiente, anche se non mancano eccezioni su particolari classi d'interesse.

3.3.1 L'espressività di ASP

Anche se questa tesi affronta il calcolo dei modelli stabili dal punto di vista sperimentale, occorre dare un inquadramento del problema dal punto di vista della teoria della complessità.

In generale, i programmi logici nella semantica dei modelli stabili sono molto espressivi, cioè, in presenza di simboli di funzione, permettono di esprimere gli insiemi iper-aritmetici [7], in altre parole permettono di descrivere funzioni correntemente non computabili in tempi accettabili.

D'altra parte, se il linguaggio viene ristretto all'ambito proposizionale, cioè non sono permessi simboli di funzione, l'espressività – e quindi la complessità – calano drasticamente. Il risultato teorico principale a cui fa riferimento il nostro lavoro è che il problema di trovare un modello stabile per un programma logico proposizionale è NP-completo.

Teorema 3.3.1. *(da [7]) Dato un programma logico π ground, il problema di decidere se π ammette un modello stabile è NP-completo.*

Dimostrazione. La dimostrazione [8] procede in due passi. La NP-hardness discende dalla riduzione di SAT (il problema della soddisfacibilità di formule booleane) al problema ASP per mezzo di una traduzione abbastanza diretta. La completezza segue per mezzo di un algoritmo non-deterministico di verifica di interpretazioni, oppure per riduzione del problema a SAT. □

In altre parole, il problema ASP è intrattabile, ma la complessità non è *astronomica* e si situa al secondo livello della gerarchia polinomiale.

La NP-completezza del problema ASP ha varie conseguenze, sia pratiche che

teoriche. Di particolare importanza è il fatto che questo risultato mette il calcolo dei modelli stabili in *competizione* diretta coi sistemi cosiddetti di *model checking* basati sulla risoluzione del problema SAT. Esistono in letteratura vari esempi d'impiego di un risolutore SAT per il calcolo dei modelli stabili di programmi (attraverso un'opportuna codifica del problema).

Capitolo 4

Semantica esplicita per DAML+OIL

Questo capitolo contiene il risultato fondamentale ed originale di questa tesi: la traduzione dei linguaggi RDF, RDFS e DAML+OIL in istanze ASP. La traduzione consiste sia in una trasformazione sintattica di asserzioni in fatti logici, ma soprattutto nella definizione di regole ASP che catturano il significato emergente dei predicati principali (`rdf:type`, `rdfs:Class`, ecc.) su cui il Web semantico verrà costruito.

4.1 Introduzione

RDF e con esso RDFS e DAML+OIL sono linguaggi atti alla creazione di metadati che permettano di ragionare sui dati contenuti in una pagina del Web semantico. Questi linguaggi, come già visto, intendono rappresentare la realtà attraverso una rete semantica, ma né il significato della rete semantica né il significato del modello RDF sono stati fin qui definiti in modo formale e condiviso. Ciò può condurre ad una diversa interpretazione della stessa rete semantica da parte degli *agenti*, comportando così una perdita di interoperabilità; presupposto fondamentale per il Web semantico.

Ogni insieme di asserzioni fatte in Resource Description Framework (RDF), RDF Schema (RDF-S), e DAML+OIL può essere tradotto in una teoria logica (espressa usando la logica dei predicati del primo ordine) i cui modelli siano logicamente equivalenti al significato inteso espresso dall'insieme delle asserzioni date. Questa operazione fa sì che sulla traduzione così ottenuta sia possibile fare dell'inferenza utilizzando gli usuali meccanismi di deduzione automatica quali Prolog ed SMODELS. Ciò permetterà di realizzare strumenti automatici di validazione e di inferenza.

Data la rete semantica di RDF/RDFS e DAML+OIL estraiamo da essa le parti *semanticamente* importanti e le traduciamo in fatti di una teoria SMODELS. Diamo poi delle regole che permettono di evincere da essi nuove informazioni, *aggiungendole* alla teoria stessa per permettere ulteriore inferenza su di loro. Concludiamo poi con una parte di validazione che permetta di verificare, nei limiti del possibile, che le asserzioni fatte siano semanticamente corrette.

La validazione e l'inferenza sono qui fatte con l'assunzione di mondo chiuso (CWA) sulle asserzioni presenti nella KB. A causa della natura distribuita del Web, il documento da validare potrebbe non contenere tutte le informazioni necessarie, ciò potrebbe causare risultati incompleti o scorretti.

Quello su cui siamo maggiormente concentrati sono i meccanismi di ereditarietà. Ecco perché non ci preoccupiamo qui di altri dettagli che non siano strettamente legati all'ereditarietà ed alla conseguente possibilità di validazione. La strada seguita è fortemente basata sulla rete semantica ed utilizza solo un numero ristretto di considerazioni che non si possono evincere dalla rete semantica stessa. Le considerazioni aggiuntive si baseranno il più delle volte sugli usuali concetti legati alle proprietà degli insiemi.

Per quanto riguarda la definizione dei domini e codomini di una proprietà esistono due scuole di pensiero. La prima è di tipo *descrittivo*, e prevede che l'aver definito il dominio di una proprietà serve per asserire che un oggetto, che riveste il ruolo di soggetto in una asserzione che ha quella proprietà come predicato, sia istanza implicita della classe che è dominio di quella proprietà. L'altro approccio è di tipo *prescrittivo*, prevede cioè di imporre che il soggetto sia stato dichiarato preventivamente istanza della classe dominio di una certa proprietà. In questa tesi abbiamo scelto un approccio di tipo prescrittivo in quanto siamo qui fortemente interessati a fornire uno strumento forte di validazione che permetta di avere un certo controllo sulle asserzioni. I due approcci, dal punto di vista dei modelli, ossia delle asserzioni vere deducibili, sono pressoché equivalenti; ecco quindi che un maggior controllo ci è parso qui più interessante da esplorare. C'è da dire che nessuno dei due approcci pare definitivo, visto che entrambi mostrano di avere delle limitazioni legate al Web semantico. Se il primo risulta più compatto, permettendo di evitare di effettuare delle istanziazioni esplicite, il secondo permette però di effettuare controlli che il primo non considera neppure. La discussione è a tutt'oggi aperta e riveste un ruolo fondamentale, trattandosi infatti della base sulla quale costruire il Web semantico.

La nostra convinzione è che per la *costruzione* del Web semantico sia di assoluta necessità una semantica non ambigua. Il fatto di esplicitarla attraverso la sua traduzione in un programma logico ci pare il modo più definitivo ed incontrovertibile per farlo. Avremo infatti che oltre all'evidenziare gli errori semantici presenti nel documento, i modelli stabili del programma rappresenteranno insiemi massimali e consistenti di fatti veri e fatti inferibili, rispetto ai fatti conosciuti ed alle regole adoperate.

4.2 La rete semantica di partenza

Consideriamo qui un grafo RDF non ambiguo e ground. Ground nel senso che tutti i nodi sono etichettati con dei nomi e non ambiguo perché tutti i nodi sono etichettati con dei nomi diversi. Questo grafo, che rappresenta unitamente gli Schemi RDF, RDFS e DAML+OIL è il nostro grafo di riferimento. Da questo estraiamo un sottografo che descrive le proprietà e le classi che noi abbiamo considerato salienti per questa trattazione. In Appendice H è riportata la rete semantica di cui andremo ad esplicitare la semantica.

4.3 Da rappresentazione a conoscenza

Il primo passo da fare è quello di passare da un livello di rappresentazione, il grafo, ad uno di conoscenza, dei fatti SMODELS. Ciò è fatto introducendo il predicato ternario `t(s,p,o)` che è vero, quindi parte dei fatti ground della teoria, se all'interno del grafo esiste un arco etichettato con `p` che abbia origine in un nodo etichettato con `s` e raggiunga un nodo etichettato con `o`.

Quelli che seguono sono tutti i fatti ground della teoria che rappresenta la rete semantica. Diremo infatti che l'esistenza nel grafo di un arco etichettato `p` che va da `s` a `o` corrisponde a considerare vera quell'asserzione RDF, qui esplicitata dal predicato `t(-,-,-)`.

```
t("rdfs:Resource",      "rdf:type",          "rdfs:Class").
t("rdf:type",           "rdf:type",          "rdf:Property").
t("rdf:type",           "rdfs:domain",       "rdfs:Resource").
t("rdf:type",           "rdfs:range",        "rdfs:Class").
t("rdfs:Class",         "rdf:type",          "rdfs:Class").
t("rdfs:Class",         "rdfs:subClassOf",   "rdfs:Resource").
```

```
t("rdfs:subClassOf",    "rdf:type",          "rdf:Property").
t("rdfs:subClassOf",    "rdfs:domain",       "rdfs:Class").
t("rdfs:subClassOf",    "rdfs:range",        "rdfs:Class").
t("rdfs:subPropertyOf", "rdf:type",          "rdf:Property").
t("rdfs:subPropertyOf", "rdfs:domain",       "rdf:Property").
t("rdfs:subPropertyOf", "rdfs:range",        "rdf:Property").
t("rdf:Property",       "rdf:type",          "rdfs:Class").
t("rdf:Property",       "rdfs:subClassOf",   "rdfs:Resource").
t("rdfs:domain",        "rdf:type",          "rdf:Property").
t("rdfs:domain",        "rdfs:domain",       "rdf:Property").
t("rdfs:domain",        "rdfs:range",        "rdfs:Class").
t("rdfs:range",         "rdf:type",          "rdf:Property").
t("rdfs:range",         "rdfs:domain",       "rdf:Property").
t("rdfs:range",         "rdfs:range",        "rdfs:Class").
t("rdf:Statement",      "rdf:type",          "rdfs:Class").
t("rdf:Statement",      "rdfs:subClassOf",   "rdfs:Resource").
t("rdf:subject",        "rdf:type",          "rdf:Property").
t("rdf:subject",        "rdfs:domain",       "rdf:Statement").
t("rdf:subject",        "rdfs:range",        "rdfs:Resource").
t("rdf:predicate",      "rdf:type",          "rdf:Property").
t("rdf:predicate",      "rdfs:domain",       "rdf:Statement").
t("rdf:predicate",      "rdfs:range",        "rdf:Property").
t("rdf:object",         "rdf:type",          "rdf:Property").
t("rdf:object",         "rdfs:domain",       "rdf:Statement").
t("daml:Class",         "rdf:type",          "rdfs:Class").
t("daml:Class",         "rdfs:subClassOf",   "rdfs:Class").
t("daml:disjointWith",  "rdf:type",          "rdf:Property").
t("daml:disjointWith",  "rdfs:domain",       "daml:Class").
t("daml:disjointWith",  "rdfs:range",        "daml:Class").
t("daml:compelmentOf",  "rdf:type",          "rdf:Property").
t("daml:compelmentOf",  "rdfs:domain",       "daml:Class").
t("daml:compelmentOf",  "rdfs:range",        "daml:Class").
t("daml:compelmentOf",  "rdfs:subPropertyOf","daml:disjointWith").
t("daml:equivalentTo",  "rdf:type",          "rdf:Property").
t("daml:equivalentTo",  "rdfs:domain",       "rdfs:Resource").
t("daml:equivalentTo",  "rdfs:range",        "rdfs:Resource").
```

```
t("daml:Transitive",    "rdf:type",           "rdfs:Class").
t("daml:Transitive",    "rdfs:subClassOf",    "rdf:Property").
t("daml:UniqueProperty","rdf:type",           "rdfs:Class").
t("daml:UniqueProperty","rdfs:subClassOf",    "rdf:Property").
```

All'interno dei fatti ground vi sono proprietà e classi RDF, RDFS e DAML+OIL. Questo perché il nostro scopo è mostrarne la loro integrazione all'interno di un unico ambiente deduttivo, fornendo al contempo uno strumento inferenziale che permetta di scrivere e validare insiemi di asserzioni DAML+OIL di piccola taglia.

4.4 Preliminari

Abbiamo tradotto la rete semantica, relativa alle definizioni di base, in un insieme di fatti ground attraverso il predicato `t(-,-,-)`. Volendo inferire nuovi fatti veri introduciamo un nuovo predicato ternario `triple(-,-,-)` che ci servirà per inferire nuova conoscenza ma che allo stesso tempo è vero su tutte le triple su cui è vero il predicato `t(-,-,-)`. In questo modo tutti i fatti del tipo `triple(s,p,o)` rappresenteranno al contempo ciò che è vero perché direttamente asserito dal modello base sia ciò che è da noi inferito.

$$\forall x, y, z(t(x, y, z) \leftrightarrow triple(x, y, z)) \tag{4.4.1}$$

SMODELS ha la necessità di *lavorare* in modo ground e quindi su di un dominio esplicito. Per fare ciò introduciamo un predicato unario `d(-)` che rappresenta il nostro dominio definito in modo tale da essere vero su tutti i *nomi* che compaiono all'interno della teoria.

$$\forall x, y, z(t(x, y, z) \leftrightarrow d(x) \wedge d(y) \wedge d(z)) \tag{4.4.2}$$

Per ragioni di compattezza, quando daremo la definizione formale della semantica, prima di quella esplicita, utilizzeremo dei predicati ausiliari. I seguenti possono anche essere visti come il caso base della definizione induttiva che daremo in seguito.
Una risorsa è tale se è istanza della classe `rdfs:Resource`.

$$\forall x(triple(x,''rdf:type'',''rdfs:Resource'') \leftrightarrow resource(x)) \tag{4.4.3}$$

Una classe è tale se è istanza della classe `rdfs:Class`.

$$\forall x(triple(x,''rdf:type'',''rdfs:Class'') \leftrightarrow class(x)) \tag{4.4.4}$$

Una proprietà è tale se è istanza della classe `rdf:Property`.

$$\forall x(triple(x,''rdf:type'',''rdf:Property'') \leftrightarrow property(x)) \tag{4.4.5}$$

La *tipizzazione* o *istanziazione* avviene attraverso il predicato `rdf:type`.

$$\forall x,y(triple(x,''rdf:type'',y) \leftrightarrow type(x,y)) \tag{4.4.6}$$

I predicati che servono per estrarre delle informazioni dalle triple rendendole esplicite e permettendoci una scrittura più compatta all'interno del testo.
Predicati RDF:

$$\begin{gathered}\forall x(triple(x,''rdf:type'',''rdf:Statement'') \leftrightarrow statement(x))\\ \forall x,y(triple(x,''rdf:subject'',y) \leftrightarrow subject(x,y))\\ \forall x,y(triple(x,''rdf:object'',y) \leftrightarrow object(x,y))\\ \forall x,y(triple(x,''rdf:predicate'',y) \leftrightarrow predicate(x,y))\end{gathered} \tag{4.4.7}$$

Predicati RDFS:

$$\begin{gathered}\forall x,y(triple(x,''rdfs:subClassOf'',y) \leftrightarrow subClassOf(x,y))\\ \forall x,y(triple(x,''rdfs:subPropertyOf'',y) \leftrightarrow subPropertyOf(x,y))\\ \forall x,y(triple(x,''rdfs:domain'',y) \leftrightarrow domain(x,y))\\ \forall x,y(triple(x,''rdfs:range'',y) \leftrightarrow range(x,y))\end{gathered} \tag{4.4.8}$$

Predicati DAML+OIL:

$$\begin{gathered}\forall x(triple(x,''rdf:type'',''daml:Transitive'') \leftrightarrow transitive(x))\\ \forall x(triple(x,''rdf:type'',''daml:UniqueProperty'') \leftrightarrow uniqueProperty(x))\\ \forall x,y(triple(x,''daml:disjointWith'',y) \leftrightarrow disjointWith(x,y))\\ \forall x,y(triple(x,''daml:complementOf'',y) \leftrightarrow complementOf(x,y))\\ \forall x,y(triple(x,''daml:equivalentTo'',y) \leftrightarrow equivalentTo(x,y))\end{gathered} \tag{4.4.9}$$

Sfruttando le definizioni di cui sopra definiamo un predicato ausiliario che ci servirà per compattezza descrittiva.

$$\begin{gathered}\forall st,x,y,z(statement(st) \wedge subject(x) \wedge predicate(y) \wedge object(z)\\ \downarrow\\ reifiedStatement(st,x,y,z))\end{gathered} \tag{4.4.10}$$

4.5 La semantica esplicita

Quella che segue è l'effettiva esplicitazione della semantica che proponiamo. Sono cioè date delle regole logiche che permettono di *derivare* ciò che all'interno della rete semantica è detto in modo implicito, rendendolo così evidente. Mostreremo infatti come, dopo il calcolo del modello del programma, la rete semantica di partenza si sarà *arricchita* di nuovi archi. Questo fatto risulterà ancor più eividente nel Capitolo 5 quando mostreremo effettivamente degli esempi.

Si fa qui l'assunto che i nomi delle risorse siano URI lecite, ciò può essere pensato come dato in fase di costruzione delle triple. Sono infatti già disponibili dei tools che traducono un documento RDF XML in insiemi di triple, che quindi possono essere pensati poi rappresentati all'interno della nostra teoria attraverso il predicato `t(-,-,-)`.

In questo momento ci preoccupiamo solo della costruzione della semantica di base; non ci preoccupiamo di come altri fatti logici verranno poi aggiunti alla teoria per essere validati. Nel Capitolo 5 vedremo due possibili soluzioni a questo problema.

Mostreremo la definizione della semantica esplicita per traduzione, considerando che il lettore abbia già letto e capito i contenuti di tutti i capitoli precedenti. Capitoli ai quali rimandiamo per informazioni specifiche relative all'uso ed al significato di ciò che qui viene utilizzato con lo scopo di definire la semantica.

Definizione 4.5.1. `rdfs:Resource`
Una risorsa è qualsiasi cosa che sia istanza di una classe che sia sottoclasse della classe `rdfs:Resource`.

$$\forall r, c(subClassOf(c,''rdfs : Resource'') \land type(r, c) \rightarrow resource(r)) \tag{4.5.1}$$

Commento 4.5.1. Questa definizione nasce dalla volontà di attenerci strettamente a ciò che è asserito nel modello RDF, senza aggiungere altre considerazioni. Sebbene siano a noi note possibili ed alternative definizioni di ciò che è una risorsa, siamo altresì convinti che una revisione di questa definizione dovrebbe prevedere da parte del W3C anche una revisione del relativo Schema RDF. Consideriamo infatti esso, unitamente alla grammatica EBNF di RDF XML, alla teoria sulle reti semantiche e alla teoria degli insiemi i nostri unici riferimenti. Asseriamo quindi che se si volesse che un letterale fosse una risorsa la classe `rdfs:Literal` dovrebbe essere dichiarata esplicitamente dal W3C come `rdfs:subClassOf` della classe `rdfs:Resource`. In considerazione del fatto che sulla *natura* stessa delle istanze della classe `rdfs:Literal` le nostre posizioni risultano essere divergenti rispetto a quelle del W3C preferiamo omettere in questa trattazione i problemi relativi ai letterali, considerando solo le risorse in senso stretto. Vi è inoltre da osservare che questo tipo di definizione fa sì

che tutte le classi e proprietà base siano effettivamente delle risorse esse stesse. Ad esempio `rdfs:Resource` è una risorsa perché essa è istanza della classe `rdfs:Class` che è sottoclasse della classe `rdfs:Resource`, anche `rdfs:Class` è una risorsa perché è istanza di sé stessa e quindi essendo sottoclasse di `rdfs:Resource` risulta essere essa stessa una risorsa.

La precedente definizione formale riscritta come programma logico SMODELS risulta essere:

```
triple(R, "rdf:type", "rdfs:Resource") :-
   d(R),
   d(C),
   triple(C, "rdf:type", "rdfs:Class"),
   triple(C, "rdfs:subClassOf", "rdfs:Resource"),
   triple(R, "rdf:type", C).
```

Definizione 4.5.2. `rdf:type`

Si dice che un oggetto è istanze di una certa classe se quest'ultima è sopraclasse di una classe di cui l'oggetto è istanza.

$$\forall x, y, c(subClassOf(c, y) \wedge type(x, c) \rightarrow type(x, y)) \tag{4.5.2}$$

Esempio 4.5.1. Questa definizione sfrutta in sé la transitività della `rdfs:subClassOf` che ci permette di dire che se `pluto` è istanza della classe `Bassotto` e la classe `Bassotto` è sottoclasse della classe `Cane` allora `pluto` è istanza della classe `Cane`.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "rdf:type", C) :-
   d(A),
   d(B),
   d(C),
   triple(A, "rdf:type", "rdfs:Resource"),
```

```
triple(C, "rdf:type", "rdfs:Class"),
triple(B, "rdf:type", "rdfs:Class"),
triple(B, "rdfs:subClassOf", C),
triple(A, "rdf:type", B).
```

Notiamo come sia sempre necessario *tipare* tutte le variabili usando l'apposito predicato `d(-)`. Anche se come vedremo in un momento successivo esistono dei predicati di *errore* che verificano che siano rispettati i domini e codomini, preferiamo qui palesare esplicitamente il fatto che la proprietà `rdf:type` ha come dominio le risorse e come codominio le classi.

Definizione 4.5.3. *Le asserzioni reificate non sono delle triple*

Abbiamo visto come sia possibile reificare una asserzione, asserzione che però non compare in nessuna tripla esplicitamente. Ci si potrebbe aspettare che per inferenza la tripla reificata fosse considerata vera, ma ciò non è così. Questo perché si presuppone che una asserzione reificata, essendo quindi predicabile, può non essere vera. Ad esempio, se reificassimo la frase *Le foglie cadono in Agosto* potremmo poi asserire che la risorsa che di essa ne è il nome è a sua volta istanza della classe delle frasi stupide. Non ha quindi senso parlare qui di verità dell'asserzione reificata[1].

Definizione 4.5.4. `rdfs:subClassOf`

Una classe A è sottoclasse della classe C se esiste una classe B che sia sottoclasse di C e che sia sopraclasse di A (*transitività*).

$$\forall x, y, z(subClassOf(x, y) \wedge subClassOf(y, z) \rightarrow subClassOf(x, z)) \qquad (4.5.3)$$

Commento 4.5.2. Non sussiste nessuna asserzione esplicita RDF che dica che la proprietà `rdfs:subClassOf` sia transitiva ma è *ragionevole* pensare che lo sia. L'ereditare

[1] Non escludiamo qui però la possibilità che in futuro si realizzino classi particolari che permettano di esplicitare il valore di verità di una asserzione reificata.

di essere sottoclasse di tutte le sopraclassi permette fra le altre cose di ereditare di essere sottoclasse di `rdfs:Resource` e quindi permettere alle proprie istanze di essere delle risorse.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "rdfs:subClassOf", C):-
   d(A),
   d(B),
   d(C),
   triple(A, "rdf:type", "rdfs:Class"),
   triple(B, "rdf:type", "rdfs:Class"),
   triple(C, "rdf:type", "rdfs:Class"),
   triple(A, "rdfs:subClassOf", B),
   triple(B, "rdfs:subClassOf", C).
```

Definizione 4.5.5. `rdfs:subPropertyOf`

Una proprietà ρ è sottoproprietà della proprietà ϕ se esiste una proprietà ψ che sia sottoproprietà di ϕ e che sia soprapropriet à di ρ (*transitività*).

$$\begin{gathered}\forall x, y, z(subPropertyOf(x, y) \wedge subPropertyOf(y, z) \\ \downarrow \\ subPropertyOf(x, z))\end{gathered} \tag{4.5.4}$$

Commento 4.5.3. Non sussiste nessuna asserzione esplicita RDF che dica che la proprietà `rdfs:subPropertyOf` sia transitiva ma è *ragionevole* pensare che lo sia. L'ereditare di essere sottoproprietà di tutte le sopraproprietà permette fra le altre cose di far ereditare a tutte le sopraproprietà ciò che è in relazione con la sottoproprietà.

Tradotto in SMODELS risulta come di seguito.

```
triple(P, "rdfs:subPropertyOf", Q) :-
   d(P),
   d(Q),
```

```
   d(R),
   triple(P, "rdf:type", "rdf:Property"),
   triple(Q, "rdf:type", "rdf:Property"),
   triple(R, "rdf:type", "rdf:Property"),
   triple(P, "rdfs:subPropertyOf", R),
   triple(R, "rdfs:subPropertyOf", Q).
```

Definizione 4.5.6. `rdfs:domain`

Una sottoproprietà eredita il dominio dalle proprietà di cui essa è sottoproprietà.

$$\forall x, y, z(subPropertyOf(x, z) \wedge domain(z, y) \rightarrow domain(x, y)) \tag{4.5.5}$$

Commento 4.5.4. Ciò farà sì che sia possibile fare controlli sul dominio in asserzioni che coinvolgano la sottoproprietà stessa come `predicate`, in considerazione del fatto che come sottoproprietà deve rispettare il dominio delle sue sopraproprietà.

Tradotto in SMODELS risulta come di seguito.

```
triple(P, "rdfs:domain", C) :-
   d(P),
   d(C),
   d(Q),
   triple(P, "rdf:type", "rdf:Property"),
   triple(C, "rdf:type", "rdfs:Class"),
   triple(Q, "rdf:type", "rdf:Property"),
   triple(P, "rdfs:subPropertyOf", Q),
   triple(Q, "rdfs:domain", C).
```

Definizione 4.5.7. `rdfs:range`

Una sottoproprietà eredita il codominio dalle proprietà di cui essa è sottoproprietà.

$$\forall x, y, z(subPropertyOf(x, z) \wedge range(z, y) \rightarrow range(x, y)) \tag{4.5.6}$$

Commento 4.5.5. Ciò farà sì che sia possibile fare controlli sul codominio in asserzioni che coinvolgano la sottoproprietà stessa come `predicate` in considerazione del fatto che come sottoproprietà deve rispettare il codominio delle sue sopraproprietà.

Tradotto in SMODELS risulta come di seguito.

```
triple(P, "rdfs:range", C) :-
   d(P),
   d(C),
   d(Q),
   triple(P, "rdf:type", "rdf:Property"),
   triple(C, "rdf:type", "rdfs:Class"),
   triple(Q, "rdf:type", "rdf:Property"),
   triple(P, "rdfs:subPropertyOf", Q),
   triple(Q, "rdfs:range", C).
```

Definizione 4.5.8. `daml:disjointWith` (I)

La proprietà `daml:disjointWith` è simmetrica.

$$\forall x, y(disjointWith(x, y) \rightarrow disjointWith(y, x)) \tag{4.5.7}$$

Commento 4.5.6. Anche se non è detto esplicitamente da nessuna asserzione RDF è *ragionevole* credere che se una classe A è disgiunta dalla classe B allora anche B è disgiunta da A.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "daml:disjointWith", B) :-
   d(A),
   d(B),
   triple(B, "daml:disjointWith", A).
```

Definizione 4.5.9. `daml:disjointWith` (II)

Una classe è disgiunta anche da tutte le sottoclassi di una classe che è a lei disgiunta.

$$\forall x, y, z(disjointWith(x, y) \wedge subClassOf(z, y) \rightarrow disjointWith(x, z)) \tag{4.5.8}$$

Commento 4.5.7. Vedremo nel prossimo capitolo un esempio in cui si vedrà che questa regola permette di determinare che anche le sottoclassi di classi disgiunte sono tra loro disgiunte.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "daml:disjointWith", B) :-
   d(A),
   d(B),
   d(C),
   triple(A, "daml:disjointWith", C),
   triple(B, "rdfs:subClassOf", C).
```

Definizione 4.5.10. *Ereditare dalle sottoproprietà*

Se due oggetti sono in relazione tramite una certa proprietà allora sono in relazione anche tramite tutte le sue soprapropriètà.

$$\forall p, q, x, y(triple(x, p, y) \wedge subPropertyOf(p, q) \rightarrow triple(x, q, y)) \tag{4.5.9}$$

Commento 4.5.8. Se pensiamo alle relazioni come a degli insiemi, avremo che una sottoproprietà sarà un sottoinsieme e quindi, tutto ciò che è contenuto in questo sottoinsieme sarà contenuto anche nel sovrainsieme cioè nella sua soprapropriètà.

Tradotto in SMODELS risulta come di seguito.

```
triple(S,Super,O) :-
   d(S),
   d(Super),
   d(O),
   d(Son),
   triple(Son, "rdfs:subPropertyOf", Super),
   triple(S, Son, O).
```

Definizione 4.5.11. `daml:equivalentTo` (I)

La proprietà `daml:equivalentTo` è simmetrica.

$$\forall x, y(equivalentTo(x, y) \rightarrow equivalentTo(y, x)) \tag{4.5.10}$$

Commento 4.5.9. Imporre la simmetria fa sì che l'ereditarietà sull'equivalenza venga trasportata in entrambi i sensi.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "daml:equivalentTo", B) :-
   d(A),
   d(B),
   triple(B, "daml:equivalentTo", A).
```

Definizione 4.5.12. `daml:equivalentTo` (II)

Se due nomi sono tra loro equivalenti allora possono comparire in ogni luogo dove compariva l'altro.

$$\forall s, p, o, q(equivalentTo(q, s) \land triple(q, p, o) \rightarrow triple(s, p, o)) \tag{4.5.11}$$

$$\forall s, p, o, q(equivalentTo(q, p) \land triple(s, q, o) \rightarrow triple(s, p, o)) \tag{4.5.12}$$

$$\forall s, p, o, q(equivalentTo(q, o) \land triple(s, p, q) \rightarrow triple(s, p, o)) \tag{4.5.13}$$

Commento 4.5.10. Questa proprietà (unitamente alle sue proprietà derivate e maggiormente specializzate) svolge un ruolo cruciale nella condivisione di ontologie. Basterà infatti aggiungere una sola asserzione di equivalenza di nomi per rendere tra loro due ontologie compatibili.

Tradotto in SMODELS risulta come di seguito.

```
triple(S, P, O):-
   d(S),
   d(P),
   d(O),
   d(Q),
   triple(Q, "daml:equivalentTo", S),
   triple(Q, P, O).
triple(S, P, O):-
```

```
   d(S),
   d(P),
   d(O),
   d(Q),
   triple(Q, "daml:equivalentTo", P),
   triple(S, Q, O).
triple(S, P, O):-
   d(S),
   d(P),
   d(O),
   d(Q),
   triple(Q, "daml:equivalentTo", O),
   triple(S, P, Q).
```

Commento 4.5.11. La proprietà `daml:equivalentTo` sarebbe transitiva, ma la sua transitività non è esplicitata causa il fatto che non è necessario farlo. Osserviamo infatti che se ρ è equivalente alla proprietà γ e γ è equivalente a ψ noi vorremmo che ρ fosse equivalente a γ. Considerando le proprietà come relazioni binarie e ricordando la definizione data della `daml:equivalentTo` si può ottenere che:

$$\forall a, b, \rho, \gamma, \psi(triple(a, \rho, b) \leftrightarrow triple(a, \gamma, b) \leftrightarrow triple(a, \psi, b)) \qquad (4.5.14)$$

e quindi concludere che ρ è equivalente a ψ, eliminando quindi la necessità di esplicitare la transitività.

Definizione 4.5.13. `daml:Transitive`

La classe `daml:Transitive` è la classe delle proprietà transitive.

$$\forall s, p, o, m(triple(s, p, m) \wedge triple(m, p, o) \wedge transitive(p) \rightarrow triple(s, p, o)) \quad (4.5.15)$$

Commento 4.5.12. Se una proprietà è istanza della classe `daml:Transitive` si deve comportare come una proprietà transitiva.

Tradotto in SMODELS risulta come di seguito.

```
triple(S,P,O) :-
   d(S),
   d(P),
   d(O),
   d(M),
   triple(P, "rdf:type", "daml:Transitive"),
   triple(S,P,M),
   triple(M,P,O).
```

Definizione 4.5.14. `daml:UniqueProperty`

La classe `daml:UniqueProperty` è la classe delle proprietà che sullo stesso `subject` possono avere un solo `object`.

$$\forall s, p, o1, o2(triple(s, p, o1) \land triple(s, p, o2) \land uniqueProperty(p) \\ \downarrow \\ equivalentTo(o1, o2)) \tag{4.5.16}$$

Commento 4.5.13. Se una proprietà è istanza della classe `daml:UniqueProperty` i nomi che compaiono come oggetto devono essere sinonimi dello stesso oggetto del dominio materiale[2].

Tradotto in SMODELS risulta come di seguito.

```
triple(O1, "daml:equivalentTo", O2) :-
   d(S),
   d(P),
   d(O1),
   d(O2),
   triple(P, "rdf:type", "daml:UniqueProperty"),
   triple(S, P, O1),
   triple(S, P, O2).
```

[2] È stato usato un approccio descrittivo e non prescrittivo.

Definizione 4.5.15. `daml:complementOf` (I)

Complementi diversi della stessa classe sono in realtà la stessa classe.

$$\begin{gathered}\forall x, y, z(complementOf(x, y) \wedge complementOf(x, z) \\ \downarrow \\ equivalentTo(y, z))\end{gathered} \tag{4.5.17}$$

Commento 4.5.14. Se due classi vengono definite come il complemento della stessa classe allora significa che sono a loro volta nomi diversi per identificare una stessa classe.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "daml:equivalentTo", B) :-
   d(A),
   d(B),
   d(C),
   triple(C, "daml:complementOf", A),
   triple(C, "daml:complementOf", B).
```

Definizione 4.5.16. `daml:complementOf` (II)

La proprietà `daml:complementOf` è simmetrica.

$$\forall x, y(complementOf(x, y) \rightarrow complementOf(y, x)) \tag{4.5.18}$$

Commento 4.5.15. Anche se non è detto esplicitamente da nessuna asserzione RDF è *ragionevole* credere che se una classe A è il complemento della classe B allora B è il complemento della classe A.

Tradotto in SMODELS risulta come di seguito.

```
triple(A, "daml:complementOf", B) :-
   d(A),
   d(B),
   triple(B, "daml:complementOf", A).
```

Definizione 4.5.17. `daml:complementOf` (III)

Se una classe è complemento di un altro segue che le due classi sono tra loro disgiunte.

$$\forall x, y(complementOf(x, y) \rightarrow disjointWith(x, y)) \tag{4.5.19}$$

Commento 4.5.16. Si realizza attraverso la definizione di sottoproprietà ed è fatto per rendere più compatti i controlli sulla proprietà `daml:complementOf` riducendoli a controlli sulla proprietà `daml:disjointWith`.

Tradotto in SMODELS risulta come di seguito.

```
t("daml:complementOf", "rdfs:subPropertyOf", "daml:disjointWith").
```

4.6 La Validazione

Abbiamo fatto inferenza sui fatti noti introducendo delle nuove triple nel modello. Quello che però è possibile fare sfruttando il motore inferenziale SMODELS è di effettuare dei controlli semantici sulle asserzioni, siano esse parte dei fatti noti o di ciò che è stato inferito. Se nella parte precedente il modello può essere incompleto causa del fatto che non si conosce tutto quello che è stato asserito, in questo caso la situazione data dalla CWA è fin peggiore. Come si potrà notare la presenza della negazione da fallimento (`not`) fa sì che ciò che non è presente nella KB sia considerato falso e quindi molto spesso potrà succedere che venga dato un messaggio di errore anche quando in realtà basterebbe solo aggiungere della nuova conoscenza. Ciò però non accade se si sta testando una intera ontologia, in cui è sottintesa la CWA.

In presenza di un errore semantico nel programma logico vengono visualizzati di messaggi di errore sfruttando i predicati d'appoggio `errore(-,...)`.

Errore 4.6.1. In una tripla possono comparire solo delle risorse

Deve essere verificato che all'interno del modello non compaiano triple in cui un qualsiasi elemento non sia una risorsa, ossia una istanza della classe `rdfs:Resource`.

$$\begin{aligned} &\forall x, p, y(triple(x, p, y) \wedge \; not \; resource(x) \rightarrow errore(x, p, y)) \\ &\forall x, p, y(triple(x, p, y) \wedge \; not \; resource(y) \rightarrow errore(x, p, y)) \\ &\forall x, p, y(triple(x, p, y) \wedge \; not \; resource(z) \rightarrow errore(x, p, y)) \end{aligned} \tag{4.6.1}$$

Commento 4.6.1. Osserviamo nuovamente il fatto che il *not* non è né la negazione classica né conoscenza esplicita negativa, ma è la negazione da fallimento, è la mancanza della tripla nella KB o della sua *dimostrazione*.

L'implementazione SMODELS risulta essere la seguente.

```
err("Nell'asserzione[",X,Y,Z,"]",X,"non è una risorsa.") :-
   triple(X, Y, Z),
   not triple(X, "rdf:type", "rdfs:Resource").
err("Nell'asserzione[",X,Y,Z,"]",Y,"non è una risorsa.") :-
   triple(X, Y, Z),
   not triple(Y, "rdf:type", "rdfs:Resource").
err("Nell'asserzione[",X,Y,Z,"]",Z,"non è una risorsa.") :-
   triple(X, Y, Z),
   not triple(Z, "rdf:type", "rdfs:Resource").
```

Errore 4.6.2. Non si possono formare cicli nelle sottoclassi

Deve essere verificato che all'interno del modello non compaiano triple in cui una classe sia sottoclasse di se stessa.

$$\forall x(subClassOf(x, x) \rightarrow errore(x)) \tag{4.6.2}$$

Commento 4.6.2. In questa implementazione è stato scelto di non ammettere cicli perché l'unico ragionevole utilizzo delle definizioni cicliche per sottoclassi è quello della

definizione di uguaglianza. In considerazione che abbiamo qui definito la proprietà `daml:equivalentTo` non ci è parso corretto l'ammettere cicli anche ricordando il fatto che una buona *tassonomia* di norma non li permette.

L'implementazione SMODELS risulta essere la seguente.

```
err("La classe",X,"è sottoclasse di se stessa.") :-
   triple(X,"rdfs:subClassOf", X).
```

Errore 4.6.3. Non si possono formare cicli nelle sottoproprietà

Deve essere verificato che all'interno del modello non compaiano triple in cui una proprietà sia sottoproprietà di se stessa.

$$\forall x(subPropertyOf(x,x) \rightarrow errore(x)) \quad (4.6.3)$$

Commento 4.6.3. In questa implementazione è stato scelto di non ammettere cicli perché l'unico ragionevole utilizzo delle definizioni cicliche per sottoproprietà è quello della definizione di uguaglianza. In considerazione del fatto che abbiamo qui definito la proprietà `daml:equivalentTo` non ci è parso corretto l'ammettere cicli, anche ricordando il fatto che una buona *tassonomia* di norma non li permette.

L'implementazione SMODELS risulta essere la seguente.

```
err("La proprietà",X,"è sottoproprietà di se stessa.") :-
   triple(X,"rdfs:subPropertyOf", X).
```

Errore 4.6.4. Si possono fare istanze solo di classi

Deve essere verificato che all'interno del modello non compaiano triple in cui un oggetto è istanza di una risorsa che non è una classe.

$$\forall x,y(type(x,y) \wedge \text{ } not\ class(y) \rightarrow errore(x,y)) \quad (4.6.4)$$

Commento 4.6.4. Non si può creare una istanza di una istanza di una classe a meno che non si stia istanziando una istanza della classe `rdfs:Class`.

L'implementazione SMODELS risulta essere la seguente.

```
err(X,"non è istanziabile perché non è una classe." ) :-
   triple(X, "rdf:type", "rdfs:Resource"),
   not triple(X, "rdf:type", "rdfs:Class"),
   triple(Y, "rdf:type", "rdfs:Class"),
   triple(X, "rdf:type", Y),
   triple(Z, "rdf:type", "rdfs:Resource"),
   triple(Z, "rdf:type", X).
```

Errore 4.6.5. Le classi devono essere sottoclassi di `rdfs:Resource`

Deve essere verificato che all'interno del modello tutte le classi siano sottoclassi della classe `rdfs:Resource`.

$$\forall x(class(x) \wedge \ not\ subClassOf(x,''rdfs:Resource'') \rightarrow errore(x)) \tag{4.6.5}$$

Commento 4.6.5. Questo controllo è fatto per garantire in *anticipo* che le *future* istanze di una classe siano effettivamente delle risorse.

L'implementazione SMODELS risulta essere la seguente.

```
err("La classe",X,"deve essere sottoclasse di rdfs:Resource.") :-
   triple(X, "rdf:type", "rdfs:Class"),
   X != "rdfs:Literal",
   X != "rdfs:Resource",
   not triple(X, "rdfs:subClassOf", "rdfs:Resource").
```

Commento 4.6.6. Abbiamo eliminato dai messaggi di errore la classe `rdfs:Literal`, perché qui non consideriamo i letterali delle risorse, e la classe `rdfs:Resource`, perché non è sottoclasse di se stessa[3].

[3]Non è stato contemplato dal W3C il caso in cui si decidesse di creare istanze della classe `rdfs:Resource` in quanto tali e non tramite altre classi.

Errore 4.6.6. Le sottoproprietà devono essere istanze di `rdf:Property`

Deve essere verificato che all'interno del modello tutte le sottoproprietà siano istanze della classe `rdf:Property`.

$$\forall x, y(subPropertyOf(x, y) \wedge \ not\ property(x) \rightarrow errore(x, y)) \tag{4.6.6}$$

Commento 4.6.7. Questo controllo è fatto per garantire che, quando una sottoproprietà viene a sua volta predicata attraverso una asserzione RDF, vengano superati i controlli di dominio e codominio.

Esempio 4.6.1. Se volessimo dire che una certa sottoproprietà è l'`rdf:predicate` di una asserzione reificata, questa deve essere istanza della classe `rdf:Property` perché la proprietà `rdf:predicate` ha come `rdfs:range` la classe `rdf:Property`.

L'implementazione SMODELS risulta essere la seguente.

```
err("La sottoproprietà",X, "deve essere una rdf:Property") :-
   triple(Y, "rdf:type", "rdf:Property"),
   triple(X, "rdfs:subPropertyOf", Y),
   not triple(X, "rdf:type", "rdf:Property").
```

Errore 4.6.7. Le sottoclassi devono essere istanze di `rdfs:Class`

Deve essere verificato che all'interno del modello tutte le sottoclassi siano istanze della classe `rdfs:Class`.

$$\forall x, y(subClassOf(x, y) \wedge \ not\ class(x) \rightarrow errore(x, y)) \tag{4.6.7}$$

Commento 4.6.8. Questo controllo è fatto per garantire che, quando si istanzia un oggetto di una sottoclasse, venga rispettato il codominio della `rdf:type`.

Esempio 4.6.2. Se volessimo istanziare un oggetto della sottoclasse `MiniVan` della classe `Van`, dovremmo farlo nel modo seguente:

```
t("mioMiniVan","rd:type","Minivan").
```

In considerazione del fatto che `rdf:type` ha come `rdfs:range` la classe `rdfs:Class` deve essere che `MiniVan` sia dichiarato a sua volta essere una classe ossia istanza della classe `rdfs:Class`.

L'implementazione SMODELS risulta essere la seguente.

```
err("La sottoclasse",X,"deve essere istanza di rdfs:Class") :-
   triple(Y, "rdf:type", "rdfs:Class"),
   triple(X, "rdfs:subClassOf", Y),
   not triple(X, "rdf:type", "rdfs:Class").
```

Errore 4.6.8. Rispetto del dominio

Deve essere verificato che all'interno del modello tutte le asserzioni rispettino il dominio della proprietà.

$$\begin{gathered} \forall y(\exists d \; domain(y,d) \rightarrow hasDomain(y)) \\ \forall x,y(\exists d \; domain(y,d) \wedge type(x,d) \rightarrow inDomain(x,y)) \end{gathered} \tag{4.6.8}$$

$$\begin{gathered} \forall s,p,o(triple(s,p,o) \wedge hasDomain(p) \wedge \; not \; inDomain(s,p) \\ \downarrow \\ errore(s,p,o)) \end{gathered} \tag{4.6.9}$$

Commento 4.6.9. Questo controllo è fatto per garantire che effettivamente tutte le asserzioni RDF, siano esse date o inferite, rispettino il dominio della proprietà con cui è etichettato il relativo arco sul grafo. Se non è specificato nessun dominio si considera che non vi sia nessun vincolo e quindi nessun errore.

L'implementazione SMODELS risulta essere la seguente.

```
err("Nell'asserzione [",X,Y,Z,"] c'è violazione di dominio.") :-
   triple(X,Y,Z),
```

```
   hasDomain(Y),
   not inDomain(X,Y).
hasDomain(Y) :-
   triple(Y, "rdfs:domain", D).
inDomain(X,Y) :-
   triple(Y, "rdfs:domain", D),
   triple(X, "rdf:type", D).
```

Commento 4.6.10. La necessità dei predicati ausiliari è dovuto al fatto che qui ci si chiede se data una tripla, in cui sulla proprietà che etichetta l'arco è stato definito almeno un dominio, il soggetto dell'asserzione ricade in almeno uno dei domini della proprietà stessa.

Errore 4.6.9. Rispetto del codominio

Deve essere verificato che all'interno del modello tutte le asserzioni rispettino il codominio della proprietà.

$$\begin{gathered} \forall y(\exists r\ range(y,r) \rightarrow hasRange(y)) \\ \forall x,y(\exists r\ range(y,r) \wedge type(x,r) \rightarrow inRange(x,y)) \end{gathered} \tag{4.6.10}$$

$$\begin{gathered} \forall s,p,o(triple(s,p,o) \wedge hasRange(p) \wedge\ not\ inRange(o,p) \\ \downarrow \\ errore(s,p,o)) \end{gathered} \tag{4.6.11}$$

Commento 4.6.11. Questo controllo è fatto per garantire che effettivamente tutte le asserzioni RDF, siano esse date o inferite, rispettino il codominio della proprietà con cui è etichettato il relativo arco sul grafo. Se non è specificato nessun codominio si considera che non vi sia nessun vincolo e quindi nessun errore.

L'implementazione SMODELS risulta essere la seguente.

```
err("Nell'asserzione [",X,Y,Z,"] c'è violazione di codominio.") :-
```

```
   triple(X,Y,Z),
   hasRange(Y),
   not inRange(Z,Y).
hasRange(Y) :-
   triple(Y, "rdfs:range", R).
inRange(Z,Y) :-
   triple(Y, "rdfs:range", R),
   triple(Z, "rdf:type", R).
```

Commento 4.6.12. La necessità dei predicati ausiliari è dovuto al fatto che qui ci si chiede se data una tripla, in cui sulla proprietà che etichetta l'arco è stato definito almeno un codominio, il soggetto dell'asserzione ricade in almeno uno dei codomini della proprietà stessa.

Errore 4.6.10. Una risorsa non può essere una classe e una proprietà

Deve essere verificato che all'interno del modello tutte le asserzioni rispettino il fatto che un oggetto sia alternativamente in modo esclusivo o una classe o una proprietà.

$$\forall x(class(x) \wedge property(x) \rightarrow errore(x)) \tag{4.6.12}$$

Commento 4.6.13. Questo è un semplice controllo che dovrebbe evidenziare errori che si potrebbero presentare in fase di *fusione* di ontologie in cui, per esempio attraverso la `daml:equivalentTo`, si potrebbero rendere *uguali* cose che non possono esserlo.

L'implementazione SMODELS risulta essere la seguente.

```
err(X," può essere una classe 'out' una proprietà.") :-
   triple(X, "rdf:type", "rdfs:Class"),
   triple(X, "rdf:type", "rdf:Property").
```

Errore 4.6.11. Una risorsa non può essere una classe e una asserzione

Deve essere verificato che all'interno del modello tutte le asserzioni rispettino il fatto

che un oggetto sia alternativamente in modo esclusivo o una classe o un'asserzione reificata.

$$\forall x(class(x) \land statement(x) \rightarrow errore(x)) \tag{4.6.13}$$

Commento 4.6.14. Questo è un semplice controllo che dovrebbe evidenziare errori che si potrebbero presentare in fase di *fusione* di ontologie in cui, per esempio attraverso la `daml:equivalentTo`, si potrebbero rendere *uguali* cose che non possono esserlo.

L'implementazione SMODELS risulta essere la seguente.

```
err(X," può essere una classe 'out' un'asserzione.") :-
   triple(X, "rdf:type", "rdfs:Class"),
   triple(X, "rdf:type", "rdf:Statement").
```

Errore 4.6.12. Una risorsa non può essere una proprietà e una asserzione

Deve essere verificato che all'interno del modello tutte le asserzioni rispettino il fatto che un oggetto sia alternativamente in modo esclusivo o una proprietà o un'asserzione reificata.

$$\forall x(property(x) \land statement(x) \rightarrow errore(x)) \tag{4.6.14}$$

Commento 4.6.15. Questo è un semplice controllo che dovrebbe evidenziare errori che si potrebbero presentare in fase di *fusione* di ontologie in cui, per esempio attraverso la `daml:equivalentTo`, si potrebbero rendere *uguali* cose che non possono esserlo.

L'implementazione SMODELS risulta essere la seguente.

```
err(X," può essere una proprietà 'out' un'asserzione.") :-
   triple(X, "rdf:type", "rdf:Property"),
   triple(X, "rdf:type", "rdf:Statement").
```

Errore 4.6.13. Un'asserzione può avere un solo soggetto

Deve essere verificato che all'interno del modello tutte le asserzioni reificate rispettino

il fatto di avere un unico soggetto.

$$\begin{gathered}\forall x, y, z(statement(x) \wedge subject(x, y) \wedge subject(x, z) \wedge z \neq y \\ \downarrow \\ errore(x, y, z))\end{gathered} \tag{4.6.15}$$

Commento 4.6.16. Non consideriamo che un'asserzione reificata possa avere più di un soggetto perché ciò potrebbe creare delle situazioni ambigue che snaturerebbero il significato stesso di asserzione.

L'implementazione SMODELS risulta essere la seguente.

```
err("L'asserzione",St,"deve avere un solo soggetto.") :-
   triple(St, "rdf:type", "rdf:Statement"),
   triple(St, "rdf:subject", X),
   triple(St, "rdf:subject", Y),
   X != Y.
```

Errore 4.6.14. Un'asserzione può avere un solo predicato

Deve essere verificato che all'interno del modello tutte le asserzioni reificate rispettino il fatto di avere un unico predicato.

$$\begin{gathered}\forall x, y, z(statement(x) \wedge predicate(x, y) \wedge predicate(x, z) \wedge z \neq y \\ \downarrow \\ errore(x, y, z))\end{gathered} \tag{4.6.16}$$

Commento 4.6.17. Non consideriamo che un'asserzione reificata possa avere più di un predicato perché ciò potrebbe creare delle situazioni ambigue che snaturerebbero il significato stesso di asserzione.

L'implementazione SMODELS risulta essere la seguente.

```
err("L'asserzione",St,"deve avere un solo predicato.") :-
```

```
   triple(St, "rdf:type", "rdf:Statement"),
   triple(St, "rdf:predicate", X),
   triple(St, "rdf:predicate", Y),
   X != Y.
```

Errore 4.6.15. Un'asserzione può avere un solo oggetto

Deve essere verificato che all'interno del modello tutte le asserzioni reificate rispettino il fatto di avere un unico oggetto.

$$\begin{gathered}\forall x, y, z(statement(x) \wedge object(x, y) \wedge object(x, z) \wedge z \neq y \\ \downarrow \\ errore(x, y, z))\end{gathered} \tag{4.6.17}$$

Commento 4.6.18. Non consideriamo che un'asserzione reificata possa avere più di un oggetto perché ciò potrebbe creare delle situazioni ambigue che snaturerebbero il significato stesso di asserzione.

L'implementazione SMODELS risulta essere la seguente.

```
err("L'asserzione",St,"deve avere un solo oggetto.") :-
   triple(St, "rdf:type", "rdf:Statement"),
   triple(St, "rdf:object", X),
   triple(St, "rdf:object", Y),
   X != Y.
```

Errore 4.6.16. Violazione di dominio nelle asserzioni reificate

Deve essere verificato che all'interno del modello in tutte le asserzioni reificate vi sia il rispetto del dominio della proprietà.

$$\begin{gathered}\forall y(\exists d\ domain(y, d) \rightarrow hasDomain(y)) \\ \forall x, y(\exists d\ domain(y, d) \wedge type(x, d) \rightarrow inDomain(x, y))\end{gathered} \tag{4.6.18}$$

$$\forall st, s, p, o(reifiedStatement(st, s, p, o) \wedge hasDomain(p) \wedge \ not\ inDomain(s, p) \\ \downarrow \\ errore(st, s, p, o)) \tag{4.6.19}$$

Commento 4.6.19. Vengono qui considerate le asserzioni reificate come delle asserzioni che, pur non essendo parte della KB perché si presume debbano essere predicate, devono comunque rispettare i vincoli di dominio.

L'implementazione SMODELS risulta essere la seguente.

```
err("Nell'asserzione ",St, "vi è una violazione di dominio.") :-
   triple(St, "rdf:type", "rdf:Statement"),
   triple(St, "rdf:predicate", P),
   triple(St, "rdf:subject", S),
   hasDomain(P),
   not inDomain(S,P).
```

Errore 4.6.17. Violazione di codominio nelle asserzioni reificate

Deve essere verificato che all'interno del modello in tutte le asserzioni reificate vi sia il rispetto del codominio della proprietà.

$$\forall y(\exists r\ range(y, r) \rightarrow hasRange(y)) \\ \forall x, y(\exists r\ range(y, r) \wedge type(x, r) \rightarrow inRange(x, y)) \tag{4.6.20}$$

$$\forall st, s, p, o(reifiedStatement(st, s, p, o) \wedge hasRange(p) \wedge \ not\ inRange(o, p) \\ \downarrow \\ errore(st, s, p, o)) \tag{4.6.21}$$

Commento 4.6.20. Vengono qui considerate le asserzioni reificate come delle asserzioni che, pur non essendo parte della KB perché si presume debbano essere predicate, devono comunque rispettare i vincoli di codominio.

L'implementazione SMODELS risulta essere la seguente.

```
err("Nell'asserzione ",St, "vi è una violazione di codominio.") :-
   triple(St, "rdf:type", "rdf:Statement"),
   triple(St, "rdf:predicate", P),
   triple(St, "rdf:object", O),
   hasRange(P),
   not inRange(O,P).
```

Errore 4.6.18. Il secondo parametro di una tripla deve essere una proprietà

Deve essere verificato che all'interno del modello, in tutte le asserzioni RDF, il secondo parametro sia un'istanza della classe `rdf:Property`.

$$\forall s, p, o(triple(s, p, o) \wedge \; not \; property(p) \rightarrow errore(s, p, o)) \tag{4.6.22}$$

Commento 4.6.21. Non è possibile etichettare un arco della rete semantica con qualche cosa che non sia una proprietà. Soggetto e oggetto della relazione devono sempre essere connessi da una proprietà, istanza quindi della classe `rdf:Property`.

L'implementazione SMODELS risulta essere la seguente.

```
err("Nell'asserzione[",X, Y, Z,"]",Y,"non è una proprietà.") :-
   triple(X, Y, Z),
   not triple(Y, "rdf:type", "rdf:Property")
```

Errore 4.6.19. Le asserzioni reificate

Deve essere verificato che all'interno del modello tutte le asserzioni reificate devono essere istanze della classe `rdfs:Resource` o sue sottoclassi.

$$\begin{gathered}\forall x, y(statement(x) \wedge type(x, y) \wedge \; not \; subClassOf(y, ''rdfs : Resource'') \\ \downarrow \\ errore(x, y))\end{gathered} \tag{4.6.23}$$

Commento 4.6.22. Non è accettato che una asserzione reificata sia istanza di classi che nulla hanno a che vedere con la classe predefinita `rdf:Statement`

L'implementazione SMODELS risulta essere la seguente.

```
err(X," , non può essere istanza di ", Y) :-
   triple(X, "rdf:type", "rdf:Statement"),
   triple(X, "rdf:type", Y),
   Y != "rdf:Statement",
   Y != "rdfs:Resource",
   not triple(Y, "rdfs:subClassOf", "rdf:Statement").
```

Commento 4.6.23. Tutto deve essere istanza della classe `rdfs:Resource` quindi non consideriamo errore il fatto che un'istanza della classe `rdf:Statement` sia anche istanza della classe `rdfs:Resource`.

Errore 4.6.20. Sottoclassi e disgiunzione

Deve essere verificato che all'interno del modello non vi siano coppie di classi disgiunte in cui una delle due è sottoclasse dell'altra.

$$\forall x, y(disjoint(x, y) \wedge subClassOf(x, y) \rightarrow errore(x, y)) \tag{4.6.24}$$

Commento 4.6.24. Non è insiemisticamente accettabile che classi disgiunte ammettano elementi comuni.

L'implementazione SMODELS risulta essere la seguente.

```
err(X,"è sottoclasse di", Y, "; sono però disgiunte.") :-
   triple(X, "rdf:type", "rdfs:Class"),
   triple(Y, "rdf:type", "rdfs:Class"),
   triple(X, "daml:disjointWith", Y),
   triple(X, "rdfs:subClassOf", Y).
```

Errore 4.6.21. Istanza di classi disgiunte

Deve essere verificato che all'interno del modello non vi siano istanze di classi tra loro

disgiunte.

$$\forall x, y, z(type(z, x) \wedge type(z, y) \wedge disjoint(x, y) \rightarrow errore(x, y, z)) \qquad (4.6.25)$$

Commento 4.6.25. Non è insiemisticamente accettabile che classi disgiunte ammettano elementi comuni.

L'implementazione SMODELS risulta essere la seguente.

```
err(Z," è istanza delle classi disgiunte", Y,Z) :-
   triple(Z, "rdf:type", X),
   triple(Z, "rdf:type", Y),
   triple(X, "daml:disjointWith", Y).
```

Commento 4.6.26. Vedremo nel Capitolo 6 come sia possibile dare una semantica diversa alla `rdf:type` ed alla `rdfs:subClassOf` in modo tale da catturare situazioni di default in cui vi siano istanze di classi tra loro disgiunte.

Errore 4.6.22. Sottoclasse di classi disgiunte

Deve essere verificato che all'interno del modello non vi siano classi che siano sottoclassi di classi tra loro disgiunte.

$$\begin{gathered}\forall x, y, z(subClassOf(x, y) \wedge subClassOf(x, z) \wedge disjoint(y, z) \\ \downarrow \\ errore(x, y, z))\end{gathered} \qquad (4.6.26)$$

Commento 4.6.27. Non è insiemisticamente accettabile che una classe sia sottoclasse di due classi fra loro disgiunte.

L'implementazione SMODELS risulta essere la seguente.

```
err(Z,"sottoclasse delle classi disgiunte",X,Y) :-
   triple(Z, "rdfs:subClassOf", X),
   triple(Z, "rdfs:subClassOf", Y),
   triple(X, "daml:disjointWith", Y).
```

Errore 4.6.23. Classe disgiunta

Deve essere verificato che all'interno del modello non vi siano classi che siano disgiunte da se stesse.

$$\forall x(disjoint(x,x) \rightarrow errore(x)) \tag{4.6.27}$$

Commento 4.6.28. Non è insiemisticamente accettabile che una classe sia disgiunta da se stessa ossia che ogni punto che vi appartiene non vi appartenga.

L'implementazione SMODELS risulta essere la seguente.

```
err("La classe",X,"non può essere disgiunta da se stessa") :-
   triple(X, "daml:disjointWith", X).
```

Capitolo 5

Sperimentazione

Questo capitolo è dedicato alla sperimentazione, con il sistema inferenziale SMODELS, del programma logico che rappresenta la semantica di RDFS/DAML+OIL.

5.1 Introduzione

Abbiamo visto nel Capitolo 4 come sia possibile attraverso un programma logico SMODELS esplicitare la semantica di classi e predicati base di RDFS/DAML+OIL, permettendo da un lato di effettuare dell'inferenza su ciò che è noto, dall'altro di effettuare dei controlli semantici all'interno delle asserzioni stesse.

Quello che vedremo adesso sono alcuni casi pratici d'uso, sia per la validazione che per l'inferenza.

5.2 Le nuove conoscenze

Quello che abbiamo a disposizione fino a questo punto è un programma logico che esplicita la semantica DAML+OIL; chiameremo questo programma π_s. Vediamo ora come si possa costruire un programma logico SMODELS π_n, costituito da fatti ground,

che una volta unito a π_s potrà essere validato rispetto alla nostra semantica, e produrre per inferenza nuova conoscenza. I modi che noi proponiamo per la costruzione di π_n sono sostanzialmente due. Il primo è quello che praticamente useremo per gli esempi, e consiste nella costruzione esplicita di π_n in base alle nostre conoscenze di dominio. Il secondo è più legato ad un uso *reale* sul Web semantico, contestuale alla validazione di ontologie. Questo modo sfrutta il principio di località e referenza.

Definizione 5.2.1. Località e referenza
Data una ontologia DAML+OIL assumiamo che per la sua validazione sia necessario il suo contenuto, e ricorsivamente ciò che da esso è referenziato attraverso il meccanismo dei namespace.

Commento 5.2.1. A questo livello di trattazione siamo convinti che la rete semantica che si viene a definire, attraverso il principio di località e referenza, sia di taglia contenuta e quindi computabile con SMODELS. La cosa importante da osservare, è che la definizione di una proprietà in una ontologia deve essere, per ragioni pratiche, ivi contenuta. Nel senso che tutto ciò che deve essere detto su di lei è lì definito. Il fatto che in un altro punto del Web semantico qualcuno abbia fatto delle asserzioni sulle nostre classi e sulle nostre proprietà non deve interessare. Chi userà la nostra ontologia, per descrivere un proprio dominio semantico, lo farà riferendosi alle definizioni raggiungibili attraverso il principio di località e referenza. Si può quindi pensare che uno strumento automatico, ricevuta in ingresso la nostra ontologia, costruisca il programma logico π_n, costituito da triple, attraverso il principio di località e referenza.

Da qui in avanti ci mettiamo nella condizione di pensare di avere due programmi disponibili, e di voler calcolare della loro unione i modelli stabili. È inteso che all'interno dei modelli stabili vi sono anche i messaggi di errore; prodotti dalla validazione

attraverso il predicato `err()`.

5.3 Esempi

Seguono una serie di esempi commentati sull'uso pratico della nostro programma logico scritto per il motore inferenziale SMODELS. Ricordiamo che l'utilizzo di SMODELS è subordinato a quello di Lparse, programma che si occupa della prima fase di *groundizzazione* e conversione in rappresentazione interna.

Esempio 5.3.1. Inferenza sulle disgiunzioni

Consideriamo ora le seguenti asserzioni RDF:

```
<daml:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
</daml:Class>
<daml:Class rdf:ID="Man">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <daml:disjointWith rdf:resource="#Woman"/>
</daml:Class>
<daml:Class rdf:ID="Woman">
  <rdfs:subClassOf rdf:resource="#Person"/>
</daml:Class>
<daml:Class rdf:ID="Girl">
  <rdfs:subClassOf rdf:resource="#Woman"/>
</daml:Class>
<daml:Class rdf:ID="Boy">
  <rdfs:subClassOf rdf:resource="#Man"/>
</daml:Class>
```

Di cui il seguente programma π_n è la rappresentazione logica attraverso le triple in esso contenute.

```
t("Person", "rdf:type", "daml:Class").
```

```
t("Person", "rdfs:subClassOf", "rdfs:Resource").
t("Man", "rdf:type", "daml:Class").
t("Man", "rdfs:subClassOf", "Person").
t("Woman", "rdf:type", "daml:Class").
t("Woman", "rdfs:subClassOf", "Person").
t("Girl", "rdf:type", "daml:Class").
t("Girl", "rdfs:subClassOf", "Woman").
t("Boy", "rdf:type", "daml:Class").
t("Boy", "rdfs:subClassOf", "Man").
t("Man", "daml:disjointWith", "Woman").
```

In figura 5.1 a pagina 153 si può osservare la relativa rete semantica. In consid-

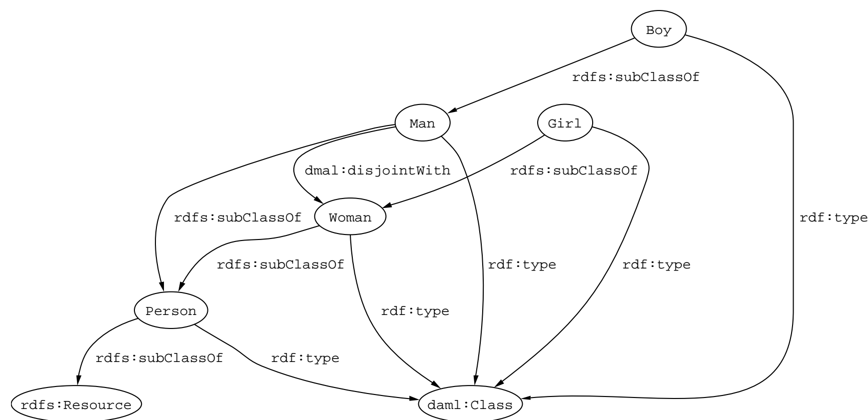


Figura 5.1: Rete semantica prima dell'inferenza [esempio: 5.3.1].

erazione del fatto che siamo qui interessati ad osservare quali conclusioni seguano dalla semantica e dalle asserzioni fatte, definiamo una query attraverso il programma SMODELS π_q:

```
true("Person",X, Y) :-
   d(X),
   d(Y),
   triple("Person",X, Y).
```

```
true("Man",X, Y) :-
   d(X),
   d(Y),
   triple("Man",X, Y).
true("Woman",X, Y) :-
   d(X),
   d(Y),
   triple("Woman",X, Y).
true("Girl",X, Y) :-
   d(X),
   d(Y),
   triple("Girl",X, Y).
true("Boy",X, Y) :-
   d(X),
   d(Y),
   triple("Boy",X, Y).
```

Vogliamo infatti conoscere tutte le triple che si possono inferire, che abbiano come soggetto le classi che abbiamo qui definito. Osserviamo che è data una gerarchia di classi, e che l'unica informazione relativa a classi disgiunte è quella che asserisce che la classe `Man` è disgiunta dalla classe `Woman`[1]. Da linea di comando invochiamo il preprocessore Lparse su $\pi_s \cup \pi_n \cup \pi_q$ messo in *pipe* con SMODELS:

```
C:\>lparse pi_s pi_n pi_q | smodels 0
```

L'output che otteniamo è il seguente:

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
true("Person","rdf:type","daml:Class")
true("Person","rdf:type","rdfs:Class")
true("Person","rdfs:subClassOf","rdfs:Resource")
```

[1] Ciò è qui considerato vero.

```
true("Person","rdf:type","rdfs:Resource")
true("Man","rdf:type","daml:Class")
true("Man","rdf:type","rdfs:Class")
true("Man","rdfs:subClassOf","rdfs:Resource")
true("Man","rdf:type","rdfs:Resource")
true("Man","daml:disjointWith","Girl")
true("Man","daml:disjointWith","Woman")
true("Man","rdfs:subClassOf","Person")
true("Woman","rdf:type","daml:Class")
true("Woman","rdf:type","rdfs:Class")
true("Woman","rdfs:subClassOf","rdfs:Resource")
true("Woman","rdf:type","rdfs:Resource")
true("Woman","daml:disjointWith","Man")
true("Woman","daml:disjointWith","Boy")
true("Woman","rdfs:subClassOf","Person")
true("Girl","rdf:type","daml:Class")
true("Girl","rdf:type","rdfs:Class")
true("Girl","rdfs:subClassOf","rdfs:Resource")
true("Girl","rdf:type","rdfs:Resource")
true("Girl","daml:disjointWith","Man")
true("Girl","daml:disjointWith","Boy")
true("Girl","rdfs:subClassOf","Woman")
true("Girl","rdfs:subClassOf","Person")
true("Boy","rdf:type","daml:Class")
true("Boy","rdf:type","rdfs:Class")
true("Boy","rdfs:subClassOf","rdfs:Resource")
true("Boy","rdf:type","rdfs:Resource")
true("Boy","rdfs:subClassOf","Man")
true("Boy","daml:disjointWith","Girl")
true("Boy","daml:disjointWith","Woman")
true("Boy","rdfs:subClassOf","Person")
```

In figura 5.2 a pagina 156 possiamo apprezzare il come l'inferenza utilizzando la semantica esplicita che abbiamo fornito, permetta di aumentare il numero di conoscenze che abbiamo sul mondo. Ricordiamo che stiamo facendo assunzione di mondo chiuso

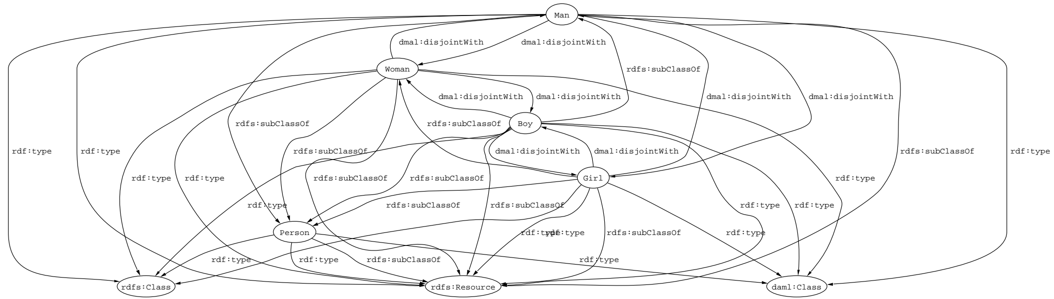


Figura 5.2: Rete semantica dopo l'inferenza [esempio: 5.3.1].

(CWA) per cui tutto ciò che non è asserito è qui considerato falso. Senza voler entrare nei dettagli diciamo che la presenza della disgiunzione tra la classe `Man` e la classe `Woman` ha permesso di inferire ben otto altre disgiunzioni tra classi.

Nell'esempio precedente abbiamo visto la struttura generale di come si opera. Affrontiamo ora dei nuovi esempi concentrandoci però solo sui modelli ottenuti come risposta alle conoscenze esplicitate attraverso il programma π_n.

Esempio 5.3.2. Errore sulla proprietà

Dato un certo insieme di asserzioni RDF questo è stato tradotto nel seguente programma π_n:

```
t("Person", "rdf:type", "rdfs:Class").
t("Person", "rdfs:subClassOf", "rdfs:Resource").
t("Franco", "rdf:type", "Person").
t("rdf:type", "Franco", "Person").
```

Possiamo osservare come erroneamente al posto di asserire che `Franco` è una istanza della classe `Person`, sia stato invertito il `subject` con il `predicate`. Calcoliamo quindi i modelli stabili[2] di $\pi_n \cup \pi_s$ ed otteniamo:

[2]I messaggi di errore sono mantenuti in lingua inglese perché sono il risultato dell'elaborazione del programma π_s che è stato scritto per essere distribuito sul Web.

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
err(">>>In the assertion[","rdf:type","Franco","Person","]",
"Franco"," is not a property.<<<")
```

La semantica di π_s ha rilevato questo errore segnalandoci in quale tripla si sia manifestato.

Esempio 5.3.3. Ereditarietà dalla sottoproprietà

Dato un certo insieme di asserzioni RDF questo è stato tradotto nel seguente programma π_n:

```
t("Person", "rdf:type", "rdfs:Class").
t("Person", "rdfs:subClassOf", "rdfs:Resource").
t("parent", "rdf:type", "rdf:Property").
t("parent", "rdfs:domain", "Person").
t("parent", "rdfs:range", "Person").
t("mother", "rdf:type", "rdf:Property").
t("mother", "rdfs:subPropertyOf", "parent").
t("franco", "rdf:type", "Person").
t("pia", "rdf:type", "Person").
t("pia", "mother", "franco").

true("pia",Y,Z):-triple("pia", Y,Z).
```

È stato asserito che `pia` è `mother` di `franco` e che `mother` è sottoproprietà della proprietà `parent`. Abbiamo introdotto la query attraverso il predicato `true()`, per poter ricavare tutte le asserzioni che hanno `pia` come `subject`. Calcoliamo quindi i modelli stabili di $\pi_n \cup \pi_s$ ed otteniamo:

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
true("pia","rdf:type","Person")
```

```
true("pia","parent","franco")
true("pia","rdf:type","rdfs:Resource")
true("pia","mother","franco")
```

La semantica di π_s ha permesso di ricavare fra le altre cose, non direttamente asserite, che `pia` è `parent` di `franco`, e questo perché è stato ereditato tramite la sottoproprietà `mother`.

Esempio 5.3.4. Violazione di dominio

Dato un certo insieme di asserzioni RDF questo è stato tradotto nel seguente programma π_n:

```
t("property", "rdf:type", "rdfs:Class").
t("property", "rdfs:subClassOf", "rdfs:Resource").
t("property", "rdfs:range", "rdf:Property").
```

Calcoliamo quindi i modelli stabili di $\pi_n \cup \pi_s$ ed otteniamo:

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
err(">>>In the assertion
[","property","rdfs:range","rdf:Property","]",
" there is a domain violation. <<<")
```

La semantica espressa da π_s ha permesso di evidenziare un errore. Non basta chiamarsi `property` per essere una `rdf:Property`. Abbiamo infatti che la `rdfs:range` ha come dominio la classe `rdf:Property`, ci si aspetta quindi che in una asserzione che coinvolga `rdfs:range` come `predicate` il `subject` sia un istanza della classe `rdf:Property`. In questo caso è stato giustamente evidenziato un errore perché `property` in realtà è stata dichiarata essere una classe. Pensiamo ora di correggere il programma asserendo che `property` è effettivamente una `rdf:Property` ottenendo:

```
t("property", "rdf:type", "rdfs:Class").
t("property", "rdfs:subClassOf", "rdfs:Resource").
t("property", "rdf:type", "rdf:Property").
t("property", "rdfs:range", "rdf:Property").
```

Calcoliamo quindi i modelli stabili del nuovo $\pi_n \cup \pi_s$ ed otteniamo:

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
err(">>>","property","is either a class or a property, not both.<<<")
```

È stato rilevato un errore causato dalla definizione simultanea di `property` come classe e come proprietà.

Esempio 5.3.5. Ciclo nella definizione delle sottoclassi

Data un certo insieme di asserzioni RDF questo è stata tradotto nel seguente programma π_n:

```
t("A", "rdf:type", "rdfs:Class").
t("A", "rdfs:subClassOf", "rdfs:Resource").
t("B", "rdf:type", "rdfs:Class").
t("B", "rdfs:subClassOf", "A").
t("B", "rdfs:subClassOf", "D").
t("C", "rdf:type", "rdfs:Class").
t("C", "rdfs:subClassOf", "B").
t("D", "rdf:type", "rdfs:Class").
t("D", "rdfs:subClassOf", "C").
```

Calcoliamo quindi i modelli stabili di $\pi_n \cup \pi_s$ ed otteniamo:

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
err(">>>The class","B","cannot be subclass of itself.<<<")
err(">>>The class","C","cannot be subclass of itself.<<<")
err(">>>The class","D","cannot be subclass of itself.<<<")
```

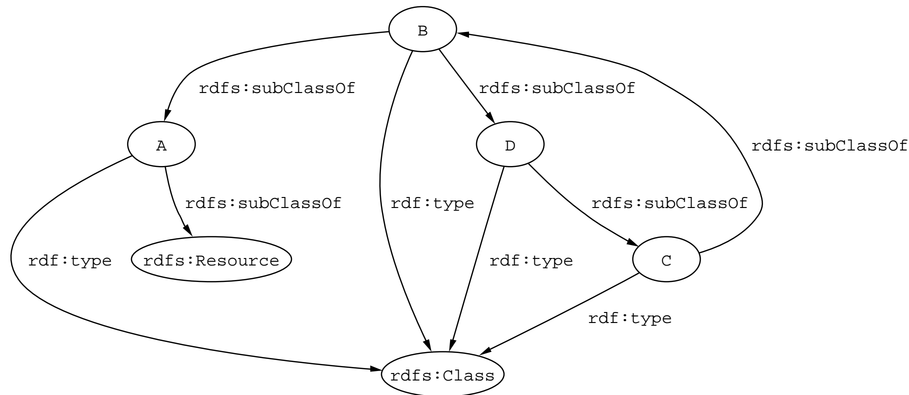


Figura 5.3: Definizione circolare delle sottoclassi [esempio: 5.3.5].

È facile osservare in figura 5.3 a pagina 160 come si sia venuto a creare un ciclo nella definizione delle sottoclassi. Questo *ciclo* è qui da considerarsi un errore, ed è quindi è stato correttamente rilevato.

Esempio 5.3.6. Nomi diversi per lo stesso oggetto

Dato un certo insieme di asserzioni RDF questo è stato tradotto nel seguente programma π_n:

```
t("Nice","rdf:type","rdfs:Class").
t("Nice","rdfs:subClassOf","rdfs:Resource").
t("Bello","rdf:type","rdfs:Class").
t("Bello","rdfs:subClassOf","rdfs:Resource").
t("diamond", "rdf:type", "Bello").
t("Bello", "daml:equivalentTo", "Nice").

true("diamond", "rdf:type", X):-triple("diamond", "rdf:type", X).
```

Calcoliamo quindi i modelli stabili di $\pi_n \cup \pi_s$ ed otteniamo:

```
smodels version 2.26. Reading...done
Answer: 1
```

```
Stable Model:
true("diamond","rdf:type","Nice")
true("diamond","rdf:type","rdfs:Resource")
true("diamond","rdf:type","Bello")
```

Si può osservare che è stata *generata* una nuova tripla, ed esattamente quella che asserisce che `diamond` è istanza della classe `Nice`. Ciò è dovuto al fatto che è stato asserito che la classe `Nice` e la classe `Bello` sono nomi diversi della stessa classe. Questo semplicissimo esempio mette in luce come, pur mancando l'UNA, sarà possibile integrare tra loro delle ontologie attraverso delle semplici asserzioni RDF che facciano uso della proprietà `daml:equivalentTo`[3].

Questi esempi ci hanno permesso di capire come attraverso un programma logico si possa ottenere al contempo, sia la possibilità di effetture la validazione (verifica formale), che l'inferenza di nuova conoscenza.

[3] In DAML+OIL sono disponibili altre proprietà per permettere l'integrazione tra ontologie (e.g. `daml:samePropertyAs`).

Capitolo 6

Estensioni per DAML+OIL

In questo capitolo descriviamo due estensioni delle definizioni DAML+OIL che ci apprestiamo a sottoporre al consorzio DAML, relative ad una definizione alternativa delle proprietà `rdf:type` e `rdfs:subClassOf`.

6.1 Introduzione

Nel Capitolo 4 abbiamo definito le proprietà `rdf:type` e `rdfs:subClassOf` attraverso un programma logico che ne esplicitava la semantica. La semantica proposta rispecchia, almeno in prima battuta, quella già nota dalla teoria degli insiemi.

Quello che faremo qui, sarà proporre una loro semantica alternativa che tenga conto della teoria dei default. Siamo infatti interessati a mostrare come sia possibile, modificando la semantica della `rdf:type` e della `rdfs:subClassOf`, fare asserzioni RDF che siano *normalmente* vere. Questo ci condurrà verso una logica non monotona in cui verranno preservate le asserzioni esplicite, ma dove alcune delle inferenze potranno essere perse nel momento in cui nuove conoscenze vengano acquisite.

Sebbene il passaggio al non monotonismo a causa della CWA sia contrario a ciò che è ragionevole assumere rispetto al Web semantico, praticamente questa ci pare

l'unica via *ragionevolmente* percorribile. Risulta infatti che molte delle conoscenze *esplicite* del Web semantico, dovranno essere inficiate per poter fornire nuove e più precise risposte. Non è pensabile infatti che in prospettiva ciò che è considerato vero oggi resti vero per sempre.

Un'altra ragione che spinge nella direzione dei default è legata alla loro capacità di esprimere, in modo compatto, conoscenze di carattere generale, anche se non sempre *vere*. Abbiamo infatti che ogni sistema di classificazione degli oggetti del *mondo* dove tener conto delle eccezioni. Anche il Web semantico dovrebbe farlo. In letteratura la principale soluzione per trattare questo tipo di situazioni è quella di modificare i meccanismi di eredità trasformandoli in default.

6.2 I default

Lo scopo delle regole di default, come vedremo, è quello di catturare i concetti di *normalmente*, di *solito* o *fino a prova contraria*; in contrapposizione all'idea del necessariamente o sempre. Ciò è fatto attraverso delle particolari regole logiche, dette appunto regole di default.

Un esempio di regola di default è la presunzione di innocenza, per cui un sospettato è innocente se è consistente assumere che non sia colpevole (ossia non vi sono prove inequivocabili di colpevolezza). Durante l'arringa non sarà quindi possibile usare come ipotesi il fatto che un imputato sia colpevole. Ciò può essere catturato dalla logica classica, ma in un modo molto complicato, attraverso la logica del secondo ordine.

Quello che faremo sarà quindi di introdurre delle regole di inferenza che diano

maggior versatilità alla già nota regola classica del modus pones:

$$\frac{\begin{array}{l}\alpha \supset \beta \\ \alpha\end{array}}{\beta} \tag{6.2.1}$$

Definizione 6.2.1. Schema di regole di default

Lo schema delle regole di default è dato da:

$$\frac{\alpha : \beta}{\gamma} \tag{6.2.2}$$

Che deve essere letto come: *se* α *è vera ed è consistente assumere che* β *sia vera allora concludiamo che* γ *è vera.* Una lettura alternativa potrebbe essere: *normalmente gli* α *sono* γ *a meno di* β*.* Queste non sono delle vere regole di inferenza, ma schemi di regole.

Commento 6.2.1. Allo schema generale è possibile associare un programma logico SMODELS:

```
gamma :- alfa, not n_beta.
:- gamma, n_gamma.
```

Dove la seconda regola rappresenta un vincolo d'integrità, cioè impone che γ e $\neg\gamma$ non siano vere contemporaneamente.

Esempio 6.2.1. Regola di default

La frase: *di solito i valdostani sono pallidi a meno che non siano maestri di sci*, può essere catturata dalla seguente regola di default:

$$\frac{valdostano : \neg maestro\ di\ sci}{pallido} \tag{6.2.3}$$

Sarà consistente assumere che un certo individuo non sia maestro di sci quando non vi sia conoscenza esplicita o dimostrabilità sul fatto che lo sia. Se ad esempio l'unica cosa

che sappiamo su Luciano è che è valdostano, possiamo inferire per default che esso sia pallido. Questa conclusione però può essere inficiata dalla scoperta che Luciano è un maestro di sci.

Definizione 6.2.2. Regole di default seminormali
Sono quelle regole di default in cui β e γ coincidono.

Esempio 6.2.2. Regola di default seminormale
Le persone di solito hanno un nome.

$$\frac{persona : nome}{nome} \tag{6.2.4}$$

Definizione 6.2.3. Credenze
Le *credenze* sono insiemi ottenuti come conclusioni da regole di default e da fatti veri. Esse sono un sovrainsieme dell'insieme di ciò che si *conosce*; insieme che rappresenta ciò che sappiamo per vero.

6.3 Pingu

Introdurremo la nostra proposta di semantica alternativa per le proprietà `rdf:type` e `rdfs:subClassOf` attraverso un esempio classico di *default inheritance* preso dalla letteratura[1].

Abbiamo detto che le regole di default sono uno strumento potente e compatto per rappresentare le nostre credenze sul mondo. È noto che *normalmente gli uccelli volano*, anche se è altresì noto che vi siano delle eccezioni, come i pinguini, che però rappresentano nella totalità delle *varietà* di uccelli casi assolutamente marginali.

[1] Problema sollevato per la prima volta in una discussione tra McCarthy e Minsky.

Pensiamo di essere interessati a descrivere l'abilità al volo delle varie specie di uccelli e pensiamo di aver definito una classe delle entità che godono della capacità di volare. Vi sono due approcci possibili, il primo consiste nell'asserire specie per specie se questa appartiene all'insieme delle entità che godono dell'abilità del volo, l'altro molto, più compatto, è dire che tutti gli uccelli volano, e di gestire separatamente le eccezioni. Consideriamo il secondo approccio il più interessante, questo in un'ottica di compattezza ed incrementabilità della conoscenza; vediamolo in dettaglio.

Grazie alla proprietà `daml:complementOf` è possibile asserire che la classe `Flying` e la classe `n_Flying` sono tra loro complementari; rispettivamente la classe delle entità che godono della proprietà del volo e quella delle entità che non ne godono. Le due classi possono quindi essere utilizzate per *implementare* una forma di negazione, nel senso che se un oggetto è istanza dell'una non può esserlo dell'altra e viceversa. Pensiamo quindi di porci nella condizione riportata in figura 6.1 a pagina 167 in cui l'asserire che la classe `Bird` è sottoclasse della classe `Flying` esprime il fatto che gli uccelli *normalmente* volano. È noto che i pinguini sono degli uccelli, per cui la classe `Penguin` viene definita come una sottoclasse della classe `Bird`. A questo punto incontriamo `pingu`, un pinguino, che quindi sarà istanza della classe `Penguin`. Se ora applichiamo l'ereditarietà come vista nel Capitolo 4, otterremo che `pingu` è istanza della classe `Flying` e quindi `pingu` gode della capacità di volare, ma attenzione, essendo un pinguino non sa volare. Pensiamo di risolvere il problema asserendo che `pingu` è istanza della classe `n_Flying`. Anche questa soluzione però porta con sé un problema, infatti, sempre in base alla semantica data nel Capitolo 4 otterremmo che `pingu` è istanza di due classi tra loro disgiunte, generando quindi un errore. Il problema che si è evidenziato è legato al fatto che l'asserire che la classe `Bird` è

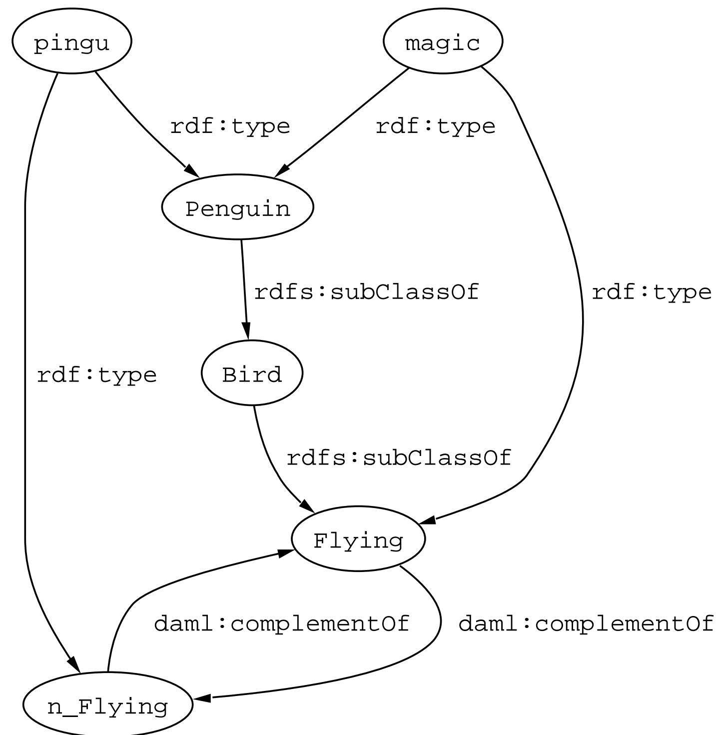


Figura 6.1: `pingu` non vola e `magic` vola.

sottoclasse della classe `Flying` è improprio, nel senso che non è vero. Esiste infatti un certo numero di istanze della classe `Bird` che non gode della capacità del volo.

L'idea che nasce a questo punto è quella di ragionare in questi termini: il fatto di conoscere in modo esplicito (dato dalla presenza di un arco nel grafo) che `pingu` non vola, ci pare qui più *forte* che l'aver ereditato che non vola attraverso due altre classi.

In questa ottica pensiamo quindi di ridefinire i programmi logici, e con essi la semantica, relativi alle due più importanti e caratterizzanti proprietà legate all'ereditarietà, la `rdf:type` e la `rdfs:subClassOf` appunto. Quello che proponiamo sono dunque due ridefinizioni che preservano l'ereditarietà già data nel Capitolo 4, ma

che in presenza di conflitti dati dalla `daml:compelmentOf` bloccano l'ereditarietà preservando gli archi espliciti, cioè quelli la cui presenza si presume rappresenti una conoscenza e non una credenza.

La rete semantica in figura 6.1 a pagina 167 rappresenta le seguenti asserzioni RDF che descrivono le nostre conoscenze sul mondo di `pingu`.

```
<daml:Class rdf:ID="Bird">
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
  <rdfs:subClassOf rdf:resource="#Flying"/>
</daml:Class>
<daml:Class rdf:ID="Penguin">
  <rdfs:subClassOf rdf:resource="#Bird"/>
</daml:Class>
<daml:Class rdf:ID="Flying">
  <daml:daml:complementOf rdf:resource="#n_Flying"/>
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
</daml:Class>
<daml:Class rdf:ID="n_Flying">
  <daml:complementOf rdf:resource="#Flying"/>
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
</daml:Class>
<Penguin rdf:ID="pingu">
  <rdf:type rdf:resource="#n_Flying"/>
</Penguin>
<Penguin rdf:ID="magic">
  <rdf:type rdf:resource="#Flying"/>
</Penguin>
```

Possiamo osservare che esiste anche `magic`, che è un pinguino magico che vola. Per questo pinguino vale lo stesso ragionamento fatto per `pingu`, ossia avendo conoscenza diretta sul fatto che vola, vorremmo che non vi fossero altre alternative possibili. Seguono i relativi fatti logici π_f derivati dalle asserzioni RDF precedenti.

```
t("Bird",       "rdfs:subClassOf",   "Flying").
```

```
t("Penguin",    "rdfs:subClassOf",   "Bird").
t("pingu",      "rdf:type",          "Penguin").
t("pingu",      "rdf:type",          "n_Flying").
t("magic",      "rdf:type",          "Penguin").
t("magic",      "rdf:type",          "Flying").
t("n_Flying",   "daml:complementOf", "Flying").
t("Flying",     "daml:complementOf", "n_Flying").
```

Esplicitiamo attraverso un programma logico π_p per SMODELS la semantica che vogliamo proporre al fine di catturare l'ereditarietà per default.

```
triple(S, "rdfs:subClassOf", O) :-
   d(S),
   d(O),
   d(B),
   d(C),
   triple(S, "rdfs:subClassOf", B),
   triple(B, "rdfs:subClassOf", O),
   not cannotBeSubClassOf(S,O).
cannotBeSubClassOf(X,C) :-
   d(X),
   d(C),
   d(A),
   triple(X, "rdfs:subClassOf", A),
   triple(A, "daml:complementOf", C).
triple(S, "rdf:type", O) :-
   d(S),
   d(C),
   d(B),
   d(O),
   triple(S, "rdf:type", B),
   triple(B, "rdfs:subClassOf", O),
   not cannotBeTypeOf(S,O).
cannotBeTypeOf(X,C) :-
   d(X),
   d(C),
   d(A),
```

```
   triple(X, "rdf:type", A),
   triple(A, "daml:complementOf", C).
```

Se ora calcoliamo i modelli stabili di $\pi_p \cup \pi_f$ otteniamo:

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model:
type("magic","Flying")
type("pingu","n_Flying")
type("magic","Penguin")
type("pingu","Penguin")
type("magic","Bird")
type("pingu","Bird")
subClassOf("Penguin","Flying")
subClassOf("Bird","Flying")
subClassOf("Penguin","Bird")
```

Quello che abbiamo ottenuto è esattamente quello che ci aspettavamo. Abbiamo infatti che `pingu` non vola perché lo avevamo come conoscenza esplicita. Questo nonostante il fatto che rispetto alla semantica usuale avrebbe dovuto sia volare che no, visto che per ereditarietà sarebbe dovuto essere istanza sia della classe `Flying` che della classe `n_Flying`. È la nuova semantica che impedisce che ciò accada. Osserviamo inoltre che giustamente la classe `Penguin` è sottoclasse della classe `Flying`. Non c'è da stupirsi del fatto che `pingu` sia un pinguino che non vola pur essendo istanza di una classe che è sottoclasse della classe delle entità che volano, è la nuova semantica che permette ciò.

Adesso modifichiamo leggermente le nostre conoscenze sul mondo di `pingu`: togliamo la conoscenza esplicita che `pingu` non vola e asseriamo che la classe `Penguin` è sottoclasse di `n_Flying`. Quello che si ottiene è la rete semantica di figura 6.2 a pagina 172 relativa alle seguenti asserzioni RDF:

```
<daml:Class rdf:ID="Bird">
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
  <rdfs:subClassOf rdf:resource="#Flying"/>
</daml:Class>
<daml:Class rdf:ID="Penguin">
  <rdfs:subClassOf rdf:resource="#Bird"/>
  <rdfs:subClassOf rdf:resource="#n_Flying"/>
</daml:Class>
<daml:Class rdf:ID="Flying">
  <daml:daml:complementOf rdf:resource="#n_Flying"/>
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
</daml:Class>
<daml:Class rdf:ID="n_Flying">
  <daml:complementOf rdf:resource="#Flying"/>
  <rdfs:subClassOf rdf:resource="rdfs:Resource"/>
</daml:Class>
<Penguin rdf:ID="pingu"/>
<Penguin rdf:ID="magic">
  <rdf:type rdf:resource="#Flying"/>
</Penguin>
```

Seguono i relativi fatti logici π_f derivati dalle asserzioni RDF precedenti.

```
t("Bird",       "rdfs:subClassOf",   "Flying").
t("Penguin",    "rdfs:subClassOf",   "Bird").
t("pingu",      "rdf:type",          "Penguin").
t("pingu",      "rdf:type",          "n_Flying").
t("magic",      "rdf:type",          "Penguin").
t("magic",      "rdf:type",          "Flying").
t("n_Flying",   "daml:complementOf", "Flying").
t("Flying",     "daml:complementOf", "n_Flying").
```

Se ora calcoliamo i modelli stabili di $\pi_p \cup \pi_f$ otteniamo:

```
smodels version 2.26. Reading...done
```

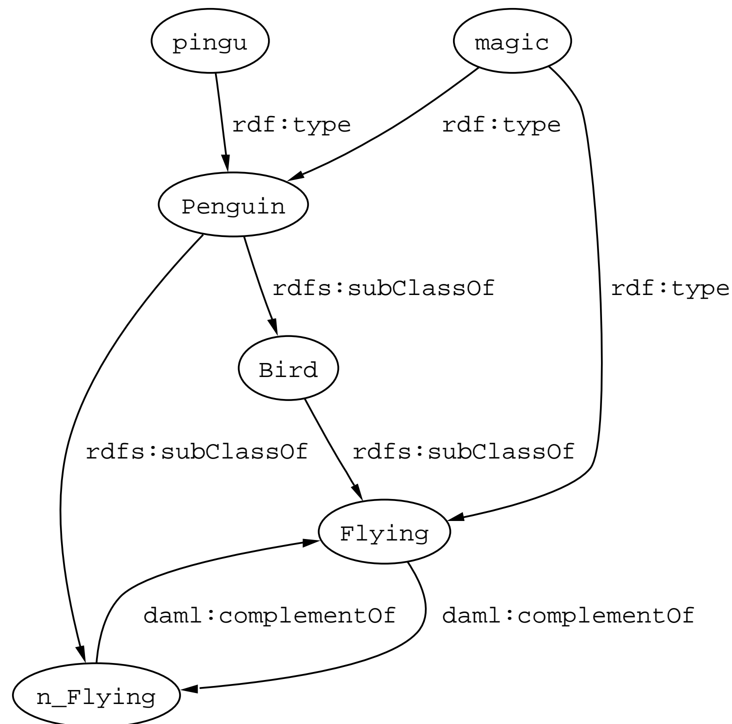


Figura 6.2: Ci sono due modelli alternativi; in uno `pingu` vola mentre nell'altro no.

```
Answer: 1
Stable Model:
type("magic","Flying")
type("pingu","Flying")
type("magic","Penguin")
type("pingu","Penguin")
type("magic","Bird")
type("pingu","Bird")
subClassOf("Bird","Flying")
subClassOf("Penguin","n_Flying")
subClassOf("Penguin","Bird")

Answer: 2
Stable Model:
```

```
type("magic","Flying")
type("pingu","n_Flying")
type("magic","Penguin")
type("pingu","Penguin")
type("magic","Bird")
type("pingu","Bird")
subClassOf("Bird","Flying")
subClassOf("Penguin","n_Flying")
subClassOf("Penguin","Bird")
```

Innanzitutto osserviamo che vi sono due modelli stabili, due modelli alternativi, due insiemi massimali e consistenti di credenze. Osserviamo inoltre che `magic` continua a volare in entrambi i modelli, e questo perché su di esso abbiamo conoscenza diretta, anche se è istanza della classe `Penguin` che è sottoclasse di `n_Flying`. Osserviamo che la classe `Penguin` non è più sottoclasse della classe `Flying`, abbiamo bloccato il default perché c'è un arco diretto con la classe `n_Flying` e quindi per incompatibilità data dalla `daml:complementOf` si è persa l'ereditarietà. La cosa ora da osservare, responsabile dei due modelli stabili, è il fatto che `pingu` ereditando di essere un uccello ci porta ad una situazione di *simmetria* semantica, come si può vedere in figura 6.3 a pagina 174. Data questa simmetria, e considerando che non c'è conoscenza esplicita sulle capacità di volo di `pingu`, è consistente assumere che `pingu` sappia volare ma anche che `pingu` non sappia volare. Grazia alla semantica data abbiamo che le due situazioni però, pur essendo entrambe plausibili, sono fra loro incompatibili. Ciò porta alla creazione di due modelli stabili che rappresentano appunto due insiemi massimamente consistenti di credenze sul mondo.

Questa semantica alternativa, partendo da una serie di asserzioni RDF, ci ha permesso di realizzare e verificare un modo attraverso il quale potrebbe essere realizzata

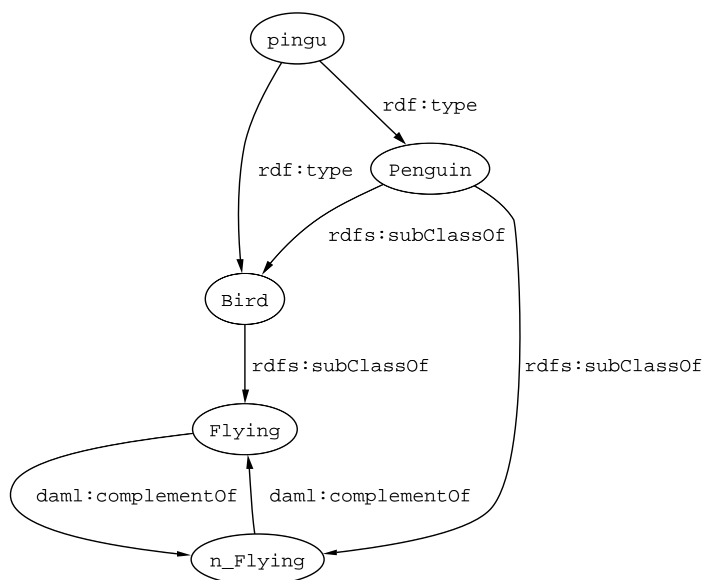


Figura 6.3: `pingu` può volare perchè eredita di essere un `Uccello`.

l'ereditarietà per default all'interno del Web semantico. In prospettiva ci pare auspicabile che meccanismi inferenziali basati su eredità per default siano parte integrante e fondamentale del Web semantico.

Conclusioni

La semantica che abbiamo proposto non è esaustiva sull'insieme delle proprietà e delle classi di DAML+OIL, ma coglie di queste quelle che noi consideriamo più significative per raggiungere un livello di sperimentabilità. Considerando infatti quello del Web semantico come un nuovo territorio da *esplorare*, ci è sembrato giusto verificare la fattibilità di questa traduzione nonché la sua effettiva efficacia. Sarà quindi possibile in futuro estendere ulteriormente questa semantica ed eventualmente modificarla. L'avere un programma logico che definisce la semantica permette di avere un documento sul quale discutere all'interno della comunità scientifica. Il grande vantaggio di ciò sta nel fatto che è possibile verificare sperimentalmente se le proprie idee sono o no conformi alle definizioni. Basterà infatti calcolare il modello stabile dell'esempio frutto del *contendere*. Se i risultati risultassero difformi da una nuova e più matura visione del Web semantico basterà modificare il programma per adeguarlo alle nuove conoscenze. Una semantica espressa attraverso un programma logico è quindi al contempo un *documento* di riferimento ed uno strumento di *sviluppo*. La semantica alternativa per `rdf:type` e `rdfs:subClassOf` mette in luce di come sia possibile modificare in modo *drammatico* il comportamento di un agente che agisse all'interno del Web semantico. Ciò sottolinea l'assoluta necessità di una semantica, definita e condivisa, dei metalinguaggi di base che servono per la creazioni di ontologie. È nostra convinzione che in futuro, al fine di raggiungere gli obiettivi che si pone il Web semantico, sarà necessario introdurre degli elementi di ragionamento non-monotono, quali quelli da noi proposti nel Capitolo 6.

Vis-a-vis le proposte oggi in elaborazione presso il consorzio W3C, la nostra semantica potrebbe venire criticata come meno *completa*. Eppure, riteniamo modestamente di aver ottenuto migliori risultati su due punti-chiave del Web semantico:

1. le proprietà che hanno nomi corrispondenti a relazioni matematiche note, come *subsetOf* sono, per semplicità, usabilità e universalità, assegnate ad una semantica standard;

2. una classificazione degli oggetti del mondo è chiaro che dovrebbe tenere conto, ed ammettere, le eccezioni. Il Web semantico dovrebbe fare altrettanto.

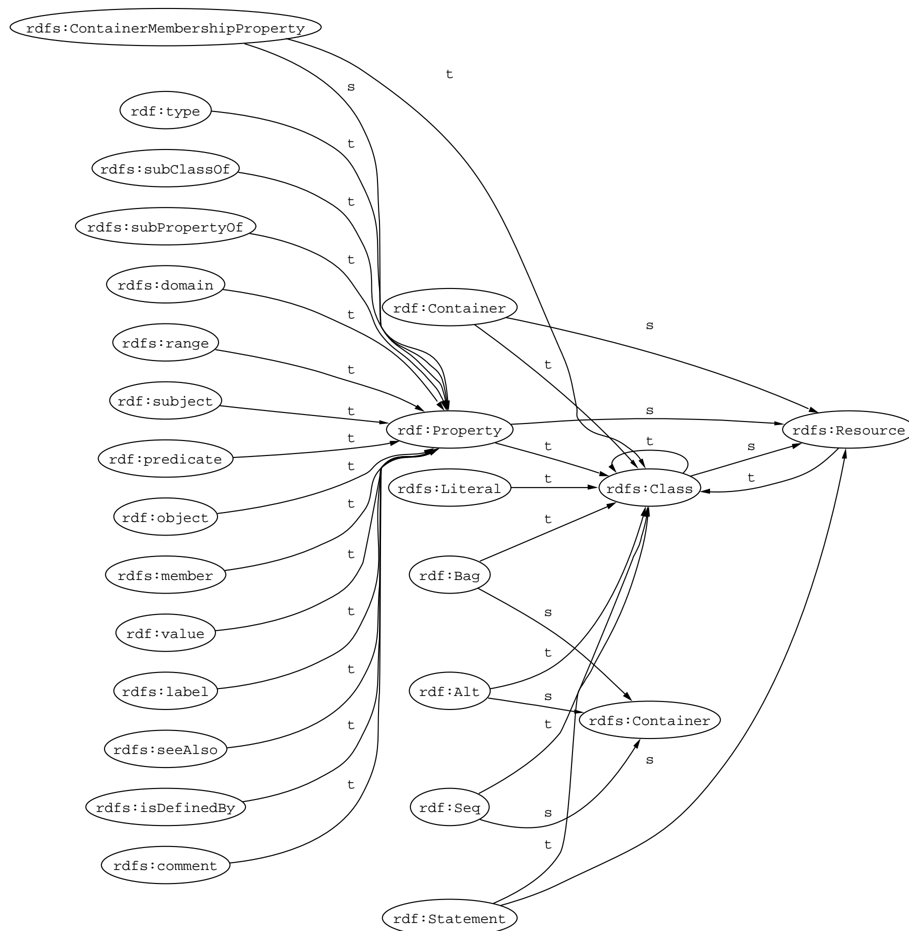
In letteratura la soluzione principale per il trattamento delle eccezioni è di rendere le regole di ereditarietà come dei default. È per questo che noi abbiamo proposto una semantica che applica l'ereditarietà per default. Riteniamo quindi che la nostra proposta di semantica sia praticamente indispensabile per realizzare gli obbiettivi del Web semantico.

Come sviluppi futuri di questo lavoro di tesi vediamo la possibilità di estendere ulteriormente i meccanismi di ereditarietà per default, inglobando in essi i risultati che in letteratura, tramite il principio di Touresky[2], permettono l'eredità di una *proprietà* dalla superclass più vicina; qualora si fossero verificati dei conflitti. I lavori di Gabaldon e Gelfond permetteranno di dare quindi una sistemazione definitiva all'ereditarietà per default all'interno del Web semantico.

[2] L'informazione più specifica soprassiede quella meno specifica.

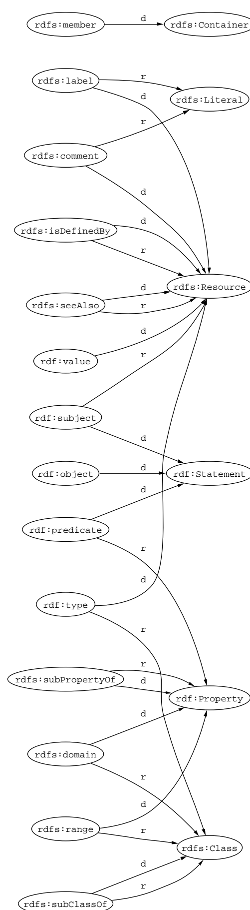
Appendice A

Rete semantica di RDF e RDFS (`t = rdf:type`, `s = rdfs:subClassOf`).



Appendice B

Rete semantica di RDF e RDFS (`d = rdfs:domain`, `r = rdfs:range`).



Appendice C

Segue l'elenco per importanza delle classi e proprietà di RDF ed RDFS.

```
rdfs:Resource .................. classe
rdf:type ....................... proprietà
rdfs:Class ..................... classe
rdfs:subClassOf ................ proprietà
rdfs:subPropertyOf ............. proprietà
rdf:Property ................... classe
rdfs:comment ................... proprietà
rdfs:label ..................... proprietà
rdfs:domain .................... proprietà
rdfs:range ..................... proprietà
rdfs:seeAlso ................... proprietà
rdfs:isDefinedBy ............... proprietà
rdfs:Literal ................... classe
rdf:Statement .................. classe
rdf:subject .................... proprietà
rdf:predicate .................. proprietà
rdf:object ..................... proprietà
rdfs:Container ................. classe
rdf:Bag ........................ classe
rdf:Seq ........................ classe
rdf:Alt ........................ classe
rdfs:ContainerMembershipProperty  classe
rdfs:member .................... proprietà
rdf:value ...................... proprietà
```

Appendice D

Lo Schema di RDF (`http://www.w3.org/1999/02/22-rdf-syntax-ns#`).

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:s="http://www.w3.org/2000/01/rdf-schema#">
<s:Class rdf:ID="Statement"
   s:comment="A triple consisting of a predicate, a subject, and an object." />
<s:Class rdf:ID="Property"
   s:comment="A name of a property, defining specific meaning for the property" />
<s:Class rdf:ID="Bag"
   s:comment="An unordered collection" />
<s:Class rdf:ID="Seq"
   s:comment="An ordered collection" />
<s:Class rdf:ID="Alt"
   s:comment="A collection of alternatives" />
<Property ID="predicate"
   s:comment="Identifies the property used in a statement when representing the
              statement in reified form">
  <s:domain rdf:resource="#Statement" />
  <s:range rdf:resource="#Property" />
</Property>
<Property ID="subject"
   s:comment="Identifies the resource that a statement is describing when representing
              the statement in reified form">
  <s:domain rdf:resource="#Statement" />
</Property>
<Property ID="object"
   s:comment="Identifies the object of a statement when representing the statement
              in reified form" />
<Property ID="type"
   s:comment="Identifies the Class of a resource" />
<Property ID="value"
   s:comment="Identifies the principal value (usually a string) of a property when the
              property value is a structured resource" />
</RDF>
```

Appendice E

Lo Schema di RDFS (`http://www.w3.org/TR/rdf-schema/`).

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Resource">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">Resource</rdfs:label>
  <rdfs:comment>The class resource, everything.</rdfs:comment>
</rdfs:Class>

<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">type</rdfs:label>
  <rdfs:comment>Indicates membership of a class</rdfs:comment>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Class">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">Class</rdfs:label>
  <rdfs:comment>The concept of Class</rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdfs:Class>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#subClassOf">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">subClassOf</rdfs:label>
  <rdfs:comment>Indicates membership of a class</rdfs:comment>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#subPropertyOf">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">subPropertyOf</rdfs:label>
  <rdfs:comment>Indicates specialization of properties</rdfs:comment>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
```

```
</rdf:Property>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">Property</rdfs:label>
  <rdfs:comment>The concept of a property.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdfs:Class>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#comment">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">comment</rdfs:label>
  <rdfs:comment>Use this for descriptions</rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#label">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">label</rdfs:label>
  <rdfs:comment>Provides a human-readable version of a resource name.</rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#domain">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">domain</rdfs:label>
  <rdfs:comment>A domain class for a property type</rdfs:comment>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#range">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">range</rdfs:label>
  <rdfs:comment>A range class for a property type</rdfs:comment>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#seeAlso">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">seeAlso</rdfs:label>
  <rdfs:comment>A resource that provides information about the subject resource</rdfs:comment>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  <rdfs:domain   rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#isDefinedBy">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
  <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#seeAlso"/>
  <rdfs:label xml:lang="en">isDefinedBy</rdfs:label>
  <rdfs:comment>Indicates the namespace of a resource</rdfs:comment>
```

```
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Literal">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">Literal</rdfs:label>
  <rdfs:comment>This represents the set of atomic values, eg. textual strings.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">Statement</rdfs:label>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  <rdfs:comment>The class of RDF statements.</rdfs:comment>
</rdfs:Class>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#subject">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">subject</rdfs:label>
  <rdfs:comment>The subject of an RDF statement.</rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">predicate</rdfs:label>
  <rdfs:comment>the predicate of an RDF statement.</rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property about="http://www.w3.org/1999/02/22-rdf-syntax-ns#object">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">object</rdfs:label>
  <rdfs:comment>The object of an RDF statement.</rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
</rdf:Property>

<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Container">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">Container</rdfs:label>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  <rdfs:comment>This represents the set Containers.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">Bag</rdfs:label>
  <rdfs:comment xml:lang="en">An unordered collection.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq">
```

```
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">Seq</rdfs:label>
  <rdfs:comment xml:lang="en">An ordered collection.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">Alt</rdfs:label>
  <rdfs:comment xml:lang="en">A collection of alternatives.</rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Container"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#ContainerMembershipProperty">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">ContainerMembershipProperty</rdfs:label>
  <rdfs:comment>
     The container membership properties, rdf:_1, rdf:_2..., all of which are
     sub-properties of 'member'.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>

<rdf:Property rdf:about="http://www.w3.org/2000/01/rdf-schema#member">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
  <rdfs:label xml:lang="en">member</rdfs:label>
  <rdfs:comment>a member of a container</rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Container"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#value">
  <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
  <rdfs:label xml:lang="en">value</rdfs:label>
  <rdfs:comment>
     Identifies the principal value (usually a string) of a property when the
     property value is a structured resource.
  </rdfs:comment>
  <rdfs:domain rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Property>

<rdf:Description rdf:about="http://www.w3.org/2000/01/rdf-schema#">
  <rdfs:seeAlso rdf:resource="http://www.w3.org/2000/01/rdf-schema-more"/>
</rdf:Description>

</rdf:RDF>
```

Appendice F

Lo Schema di DAML+OIL (`http://www.daml.org/2001/03/daml+oil#`).

```
<!-- $Revision: 1.7 $ of $Date: 2001/06/06 01:38:21 $. -->

<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns     ="http://www.daml.org/2001/03/daml+oil#"
>

<rdf:Description rdf:about="">
  <versionInfo>$Id: daml+oil.daml,v 1.7 2001/06/06 01:38:21 mdean Exp $</versionInfo>
  <imports rdf:resource="http://www.w3.org/2000/01/rdf-schema"/>
</rdf:Description>

<!-- (meta) classes of "object" and datatype classes  -->

<rdfs:Class rdf:ID="Class">
  <rdfs:label>Class</rdfs:label>
  <rdfs:comment>
    The class of all "object" classes
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Datatype">
  <rdfs:label>Datatype</rdfs:label>
  <rdfs:comment>
    The class of all datatype classes
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdfs:Class>

<!-- Pre-defined top/bottom thing/nothing most/least-general (object) classes. -->

<Class rdf:ID="Thing">
  <rdfs:label>Thing</rdfs:label>
  <rdfs:comment>
    The most general (object) class in DAML.
    This is equal to the union of any class and its complement.
```

```
  </rdfs:comment>
  <unionOf rdf:parseType="daml:collection">
    <rdfs:Class rdf:about="#Nothing"/>
    <rdfs:Class>
      <complementOf rdf:resource="#Nothing"/>
    </rdfs:Class>
  </unionOf>
</Class>

<Class rdf:ID="Nothing">
  <rdfs:label>Nothing</rdfs:label>
  <rdfs:comment>the class with no things in it.</rdfs:comment>
  <complementOf rdf:resource="#Thing"/>
</Class>

<!-- Terms for building classes from other classes. -->

<Property rdf:ID="equivalentTo"> <!-- equals? equiv? renames? -->
  <rdfs:label>equivalentTo</rdfs:label>
  <comment>
    for equivalentTo(X, Y), read X is an equivalent term to Y.
  </comment>
</Property>

<Property rdf:ID="sameClassAs">
  <rdfs:label>sameClassAs</rdfs:label>
  <comment>
    for sameClassAs(X, Y), read X is an equivalent class to Y.
    cf OIL Equivalent
  </comment>
  <rdfs:subPropertyOf rdf:resource="#equivalentTo"/>
  <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#subClassOf"/>
  <rdfs:domain rdf:resource="#Class"/>
  <rdfs:range rdf:resource="#Class"/>
</Property>

<Property rdf:ID="samePropertyAs">
  <rdfs:label>samePropertyAs</rdfs:label>
  <rdfs:comment>
    for samePropertyAs(P, R), read P is an equivalent property to R.
  </rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#equivalentTo"/>
  <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#subPropertyOf"/>
</Property>

<Property rdf:ID="sameIndividualAs">
  <rdfs:label>sameIndividualAs</rdfs:label>
  <rdfs:comment>
    for sameIndividualAs(a, b), read a is the same individual as b.
  </rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#equivalentTo"/>
  <rdfs:domain rdf:resource="#Thing"/>
  <rdfs:range rdf:resource="#Thing"/>
</Property>
```

```
<rdf:Property rdf:ID="disjointWith">
  <rdfs:label>disjointWith</rdfs:label>
  <rdfs:comment>
    for disjointWith(X, Y) read: X and Y have no members in common.
    cf OIL Disjoint
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Class"/>
  <rdfs:range rdf:resource="#Class"/>
</rdf:Property>

<Property rdf:ID="differentIndividualFrom">
  <rdfs:label>differentIndividualFrom</rdfs:label>
  <rdfs:comment>
    for differentIndividualFrom(a, b), read a is not the same individual as b.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Thing"/>
  <rdfs:range rdf:resource="#Thing"/>
</Property>


<!-- NOTE: the Disjoint class has been deleted: use disjointWith -->
<!-- or disjointUnionOf instead. -->

<rdf:Property rdf:ID="unionOf">
  <rdfs:label>unionOf</rdfs:label>
  <rdfs:comment>
    for unionOf(X, Y) read: X is the union of the classes in the list Y;
    i.e. if something is in any of the classes in Y, it's in X, and vice versa.
    cf OIL OR
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Class"/>
  <rdfs:range rdf:resource="#List"/>
</rdf:Property>

<rdf:Property rdf:ID="disjointUnionOf">
  <rdfs:label>disjointUnionOf</rdfs:label>
  <rdfs:comment>
    for disjointUnionOf(X, Y) read: X is the disjoint union of the classes in
    the list Y: (a) for any c1 and c2 in Y, disjointWith(c1, c2),
    and (b) unionOf(X, Y). i.e. if something is in any of the classes in Y, it's
    in X, and vice versa.
    cf OIL disjoint-covered
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Class"/>
  <rdfs:range rdf:resource="#List"/>
</rdf:Property>

<rdf:Property rdf:ID="intersectionOf">
  <rdfs:label>intersectionOf</rdfs:label>
  <rdfs:comment>
    for intersectionOf(X, Y) read: X is the intersection of the classes in the list Y;
    i.e. if something is in all the classes in Y, then it's in X, and vice versa.
    cf OIL AND
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Class"/>
```

```
  <rdfs:range rdf:resource="#List"/>
</rdf:Property>

<rdf:Property rdf:ID="complementOf">
  <rdfs:label>complementOf</rdfs:label>
  <rdfs:comment>
    for complementOf(X, Y) read: X is the complement of Y; if something is in Y,
    then it's not in X, and vice versa.
    cf OIL NOT
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Class"/>
  <rdfs:range rdf:resource="#Class"/>
</rdf:Property>

<!-- Term for building classes by enumerating their elements -->

<rdf:Property rdf:ID="oneOf">
  <rdfs:label>oneOf</rdfs:label>
  <rdfs:comment>
     for oneOf(C, L) read everything in C is one of the
     things in L;
     This lets us define classes by enumerating the members.
     cf OIL OneOf
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Class"/>
  <rdfs:range rdf:resource="#List"/>
</rdf:Property>

<!-- Terms for building classes by restricting their properties. -->

<rdfs:Class rdf:ID="Restriction">
  <rdfs:label>Restriction</rdfs:label>
  <rdfs:comment>
    something is in the class R if it satisfies the attached restrictions,
    and vice versa.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Class"/>
</rdfs:Class>

<rdf:Property rdf:ID="onProperty">
  <rdfs:label>onProperty</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P), read:
    R is a restricted with respect to property P.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Property>

<rdf:Property rdf:ID="toClass">
  <rdfs:label>toClass</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P) and toClass(R, X), read:
    i is in class R if and only if for all j, P(i, j) implies type(j, X).
    cf OIL ValueType
```

```
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Property>

<rdf:Property rdf:ID="hasValue">
  <rdfs:label>hasValue</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P) and hasValue(R, V), read:
    i is in class R if and only if P(i, V).
    cf OIL HasFiller
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
</rdf:Property>

<rdf:Property rdf:ID="hasClass">
  <rdfs:label>hasClass</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P) and hasClass(R, X), read:
    i is in class R if and only if for some j, P(i, j) and type(j, X).
    cf OIL HasValue
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Property>

<!-- Note that cardinality restrictions on transitive properties, or     -->
<!-- properties with transitive sub-properties, compromise decidability. -->

<rdf:Property rdf:ID="minCardinality">
  <rdfs:label>minCardinality</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P) and minCardinality(R, n), read:
    i is in class R if and only if there are at least n distinct j with P(i, j).
    cf OIL MinCardinality
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</rdf:Property>

<rdf:Property rdf:ID="maxCardinality">
  <rdfs:label>maxCardinality</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P) and maxCardinality(R, n), read:
    i is in class R if and only if there are at most n distinct j with P(i, j).
    cf OIL MaxCardinality
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</rdf:Property>

<rdf:Property rdf:ID="cardinality">
  <rdfs:label>cardinality</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P) and cardinality(R, n), read:
```

```
    i is in class R if and only if there are exactly n distinct j with P(i, j).
    cf OIL Cardinality
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</rdf:Property>

<rdf:Property rdf:ID="hasClassQ">
  <rdfs:label>hasClassQ</rdfs:label>
  <rdfs:comment>
    property for specifying class restriction with cardinalityQ constraints
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Property>

<rdf:Property rdf:ID="minCardinalityQ">
  <rdfs:label>minCardinality</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P), minCardinalityQ(R, n) and hasClassQ(R, X), read:
    i is in class R if and only if there are at least n distinct j with P(i, j)
    and type(j, X).
    cf OIL MinCardinality
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</rdf:Property>

<rdf:Property rdf:ID="maxCardinalityQ">
  <rdfs:label>maxCardinality</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P), maxCardinalityQ(R, n) and hasClassQ(R, X), read:
    i is in class R if and only if there are at most n distinct j with P(i, j)
    and type(j, X).
    cf OIL MaxCardinality
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</rdf:Property>

<rdf:Property rdf:ID="cardinalityQ">
  <rdfs:label>cardinality</rdfs:label>
  <rdfs:comment>
    for onProperty(R, P), cardinalityQ(R, n) and hasClassQ(R, X), read:
    i is in class R if and only if there are exactly n distinct j with P(i, j)
    and type(j, X).
    cf OIL Cardinality
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Restriction"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</rdf:Property>

<!-- Classes and Properties for different kinds of Property -->

<rdfs:Class rdf:ID="ObjectProperty">
```

```
  <rdfs:label>ObjectProperty</rdfs:label>
  <rdfs:comment>
    if P is an ObjectProperty, and P(x, y), then y is an object.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>

<rdfs:Class rdf:ID="DatatypeProperty">
  <rdfs:label>DatatypeProperty</rdfs:label>
  <rdfs:comment>
    if P is a DatatypeProperty, and P(x, y), then y is a data value.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>

<rdf:Property rdf:ID="inverseOf">
  <rdfs:label>inverseOf</rdfs:label>
  <rdfs:comment>
    for inverseOf(R, S) read: R is the inverse of S; i.e.
    if R(x, y) then S(y, x) and vice versa.
    cf OIL inverseRelationOf
  </rdfs:comment>
  <rdfs:domain rdf:resource="#ObjectProperty"/>
  <rdfs:range rdf:resource="#ObjectProperty"/>
</rdf:Property>

<rdfs:Class rdf:ID="TransitiveProperty">
  <rdfs:label>TransitiveProperty</rdfs:label>
  <rdfs:comment>
    if P is a TransitiveProperty, then if P(x, y) and P(y, z) then P(x, z).
    cf OIL TransitiveProperty.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#ObjectProperty"/>
</rdfs:Class>

<rdfs:Class rdf:ID="UniqueProperty">
  <rdfs:label>UniqueProperty</rdfs:label>
  <rdfs:comment>
    compare with maxCardinality=1; e.g. integer successor:
    if P is a UniqueProperty, then if P(x, y) and P(x, z) then y=z.
    cf OIL FunctionalProperty.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>

<rdfs:Class rdf:ID="UnambiguousProperty">
  <rdfs:label>UnambiguousProperty</rdfs:label>
  <rdfs:comment>
    if P is an UnambiguousProperty, then if P(x, y) and P(z, y) then x=z.
    aka injective. e.g. if firstBorne(m, Susan)
    and firstBorne(n, Susan) then m and n are the same.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#ObjectProperty"/>
</rdfs:Class>
```

```
<!-- List terminology. -->

<rdfs:Class rdf:ID="List">
  <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq"/>
</rdfs:Class>

<List rdf:ID="nil">
  <rdfs:comment>
     the empty list; this used to be called Empty.
  </rdfs:comment>
</List>

<rdf:Property rdf:ID="first">
  <rdfs:domain rdf:resource="#List"/>
</rdf:Property>

<rdf:Property rdf:ID="rest">
  <rdfs:domain rdf:resource="#List"/>
  <rdfs:range rdf:resource="#List"/>
</rdf:Property>

<rdf:Property rdf:ID="item">
  <rdfs:comment>
    for item(L, I) read: I is an item in L; either first(L, I)
    or item(R, I) where rest(L, R).
  </rdfs:comment>
  <rdfs:domain rdf:resource="#List"/>
</rdf:Property>

<!-- A class for ontologies themselves... -->

<rdfs:Class rdf:ID="Ontology">
  <rdfs:label>Ontology</rdfs:label>
  <rdfs:comment>
    An Ontology is a document that describes
    a vocabulary of terms for communication between
    (human and) automated agents.
  </rdfs:comment>
</rdfs:Class>

<rdf:Property rdf:ID="versionInfo">
  <rdfs:label>versionInfo</rdfs:label>
  <rdfs:comment>
    generally, a string giving information about this
    version; e.g. RCS/CVS keywords
  </rdfs:comment>
</rdf:Property>

<!-- Importing, i.e. assertion by reference -->

<rdf:Property rdf:ID="imports">
  <rdfs:label>imports</rdfs:label>
  <rdfs:comment>
    for imports(X, Y) read: X imports Y;
    i.e. X asserts the* contents of Y by reference;
```

```
    i.e. if imports(X, Y) and you believe X and Y says something,
    then you should believe it.
    Note: "the contents" is, in the general case,
    an il-formed definite description. Different
    interactions with a resource may expose contents
    that vary with time, data format, preferred language,
    requestor credentials, etc. So for "the contents",
    read "any contents".
  </rdfs:comment>
</rdf:Property>

<!-- Importing terms from RDF/RDFS -->

<!-- first, assert the contents of the RDF schema by reference -->
<Ontology rdf:about="">
  <imports rdf:resource="http://www.w3.org/2000/01/rdf-schema"/>
</Ontology>

<rdf:Property rdf:ID="subPropertyOf">
  <samePropertyAs  rdf:resource="http://www.w3.org/2000/01/rdf-schema#subPropertyOf"/>
</rdf:Property>

<rdfs:Class rdf:ID="Literal">
  <sameClassAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Property">
  <sameClassAs rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdfs:Class>

<rdf:Property rdf:ID="type">
  <samePropertyAs rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"/>
</rdf:Property>

<rdf:Property rdf:ID="value">
  <samePropertyAs rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#value"/>
</rdf:Property>

<rdf:Property rdf:ID="subClassOf">
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#subClassOf"/>
</rdf:Property>

<rdf:Property rdf:ID="domain">
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#domain"/>
</rdf:Property>

<rdf:Property rdf:ID="range">
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#range"/>
</rdf:Property>

<rdf:Property rdf:ID="label">
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#label"/>
</rdf:Property>

<rdf:Property rdf:ID="comment">
```

```
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#comment"/>
</rdf:Property>

<rdf:Property rdf:ID="seeAlso">
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#seeAlso"/>
</rdf:Property>

<rdf:Property rdf:ID="isDefinedBy">
  <samePropertyAs rdf:resource="http://www.w3.org/2000/01/rdf-schema#isDefinedBy"/>
  <rdfs:subPropertyOf rdf:resource="#seeAlso"/>
</rdf:Property>

</rdf:RDF>
```

Appendice G

Segue l'elenco per importanza delle classi e proprietà di DAML+OIL.

daml:Class .... classe
daml:Datatype .... classe
daml:Thing .... classe
daml:Nothing .... classe
daml:equivalentTo .... proprietà
daml:sameClassAs .... proprietà
daml:samePropertyAs .... proprietà
daml:sameIndividualAs .... proprietà
daml:disjointWith .... proprietà
daml:differentIndividualFrom .... proprietà
daml:unionOf .... proprietà
daml:disjointUnionOf .... proprietà
daml:intersectionOf .... proprietà
daml:complementOf .... proprietà
daml:oneOf .... proprietà
daml:Restriction .... classe
daml:onProperty .... proprietà
daml:toClass .... proprietà
daml:hasValue .... proprietà
daml:hasClass .... proprietà
daml:minCardinality .... proprietà
daml:maxCardinality .... proprietà
daml:cardinality .... proprietà
daml:hasClassQ .... proprietà
daml:minCardinalityQ .... proprietà
daml:maxCardinalityQ .... proprietà

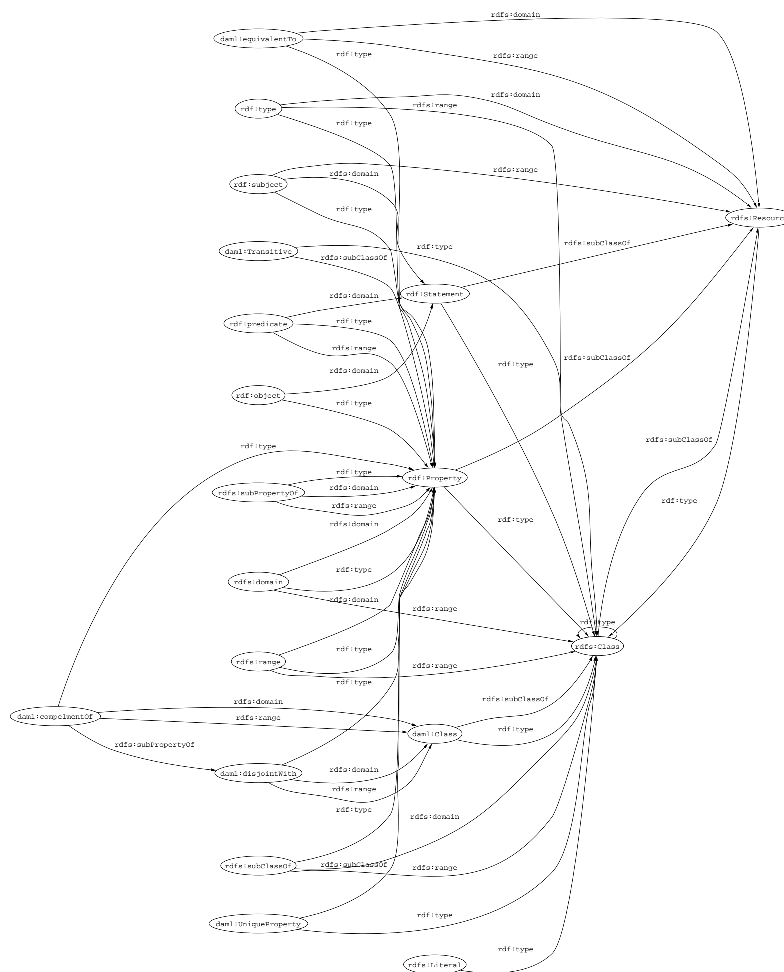
daml:cardinalityQ .... proprietà
daml:ObjectProperty .... classe
daml:DatatypeProperty .... classe
daml:inverseOf .... proprietà
daml:TransitiveProperty .... classe
daml:UniqueProperty .... classe
daml:UnambiguousProperty .... classe
daml:List .... classe
daml:nil .... daml:List
daml:first .... proprietà
daml:rest .... proprietà
daml:item .... proprietà
daml:Ontology .... classe
daml:versionInfo .... proprietà
daml:imports .... proprietà
daml:subPropertyOf .... proprietà

Classi e proprietà RDFS e RDF rese equivalenti in DAML+OIL.

daml:Literal .... classe
daml:Property .... classe
daml:type .... proprietà
daml:value .... proprietà
daml:subClassOf .... proprietà
daml:domain .... proprietà
daml:range .... proprietà
daml:label .... proprietà
daml:comment .... proprietà
daml:seeAlso .... proprietà
daml:isDefinedBy .... proprietà

Appendice H

Rete semantica RDF/RDFS/DAML+OIL esplicitata tramite SMODELS.



Glossario

In considerazione del fatto che durante la trattazione sia necessario, per brevità, l'utilizzo di acronimi segue una lista di quelli più importanti con i relativi significati.

Definizioni

AI .... Artificial Intelligence
ANSI ...... American National Standards Institute
CERN ...... Centro Europeo per la Ricerca Nucleare
CWA .... Closed World Assumption
DAML ...... DARPA Agent Mark-up Language
DARPA ..... Defense Advanced Research Projects Agency
DATALOG ... Logic Based Data Model
DNS .... Domain Name Server
DTD .... Document Type Definition
EBNF ...... Extended Backus-Naur Form
FTP .... File Transfer Protocol
GML .... Generalized Makup Language
HTML ...... HyperText Markup Language
http ...... hyper text transfer protocol
IBM .... International Business Machine
IP .... Internet Protocol
KB .... Knowledge Base
KR .... Knowlwdge Representation
KSL .... Knowledge Systems Laboratory (Stanford University)
LCW .... Localized Closed World
MIT .... Massachusetts Institute of Technology

OIL .... Ontology Inference Layer
RDF .... Resource Description Framework
RDFS ...... RDF Schema Language
RTF .... Rich Text Format
SGML ...... Standard Generalized Makup Language
UNA .... Unique Name Assumption
URI .... Universal Resource Identifier
URL .... Universal Resource Locator
W3C .... World Wide Web Consortium
WWW .... World Wide Web
XML .... eXtensible Mark-up Language
XSL .... Extensible Stylesheet Language

Risorse

Segue l'elenco delle risorse utilizzate per sviluppare questa tesi.

- `www.daml.org`
  home page del consorzio DAML+OIL;
- `www.w3c.org`
  home page del consorzio W3C;
- `http://www.daml.org/services/`
  home page di DAML-S;
- `http://www.silab.dsi.unimi.it/~fs384576/home_sw.html`
  la mia pagina personale dedicata al Web semantico;
- `http://www.ksl.stanford.edu/projects/DAML/lessons-learned.shtml`
  idee per l'estensione dell'ontologia DAML+OIL alla logica;
- `http://www.isi.edu/webscripter/`
  progetto di editor per ontologie;
- `http://www.daml.org/2001/04/hyperdaml/`
  partendo dalla URL di una ontologia o un testo marcato con RDF permette di navigare le ontologie in rete come se fossero degli ipertesti;
- `http://plucky.teknowledge.com/daml/damlquery.jsp`
  DAML Semantic Search;

- http://www.w3.org/TR/rdf-schema/
  RDF Vocabulary Description Language 1.0: RDF Schema;
- http://www.w3.org/TR/REC-rdf-syntax/
  Resource Description Framework (RDF) Model and Syntax Specification;
- http://www.w3.org/RDF/Validator/
  RDF Validation Service;
- http://www.xml.com/pub/a/2001/04/25/prologrdf/index.html
  An Introduction to Prolog and RDF;
- http://triple.semanticweb.org/
  TRIPLE Homepage;
- http://www.w3.org/RDF/Metalog/
  Metalog - The RDF querying system;
- http://www.w3.org/TR/REC-xml
  Extensible Markup Language (XML) 1.0 (Second Edition);
- http://www.w3.org/TR/REC-xml-names/
  Namespaces in XML;
- http://www.semanticweb.org/
  The Semantic Web Community Portal;
- http://www.w3.org/2001/sw/
  Semantic Web;
- http://www.w3.org/DesignIssues/UI.html#OhYeah
  The Oh yeah? button;
- http://www.isi.edu/in-notes/rfc2396.txt
  Uniform Resource Identifiers (URI): Generic Syntax;

- http://www.w3.org/Addressing/
  Naming and Addressing: URIs, URLs, ...;
- http://www.dfki.uni-kl.de/ boley/
  Home page di Harold Boley;
- http://www.daml.org/validator/
  DAML Validator;
- http://www.cs.umd.edu/projects/plus/SHOE/pubs/techrpt99.pdf
  SHOE: A Knowledge Representation Language for Internet Applications;
- http://dublincore.org/
  Dubline Core Metadata Initiative;
- http://www.daml.org/crawler/
  spider per il Web semantico;
- http://www.swi-prolog.org/packages/rdf2pl.html
  SWI-Prolog RDF parser;

Bibliografia

[1] Tim Berners-Lee, James Hendler, and Ora Lassila, *Il web semantico*, Le Scienze (2001), 77–84.

[2] Harold Boley, *Relationships between logic programming and rdf*, Proc. of PRIIA 2000, in conjunction with PRICAI 2000, Melbourne, 2000.

[3] Richard Fikes, Deborah McGuinness, and Sheila McIlraith, *Knowledge systems laboratory at stanford university*, http://www.ksl.stanford.edu, 2002.

[4] A. Van Gelder, K.A. Ross, and J. Schlipf, *The well-founded semantics for general logic programs*, Journal of the ACM, Vol. 38, 1990.

[5] M. Gelfond and V. Lifschitz, *The stable model semantics for logic programming*, Proc. of the 8th International Workshop on Non-Monotonic Reasoning, 1988.

[6] ———, *Classical negation in logic programs and disjunctive databases*, New Generation Computing (1991), pp. 365–387.

[7] W. Marek and M. Truszczyński, *Stable models an alternative logic programming paradigm*, The Journal of Logic Programming, 1999.

[8] I. Niemelä and P. Simons, *Logic programs with stable model semantics as a constraint programming paradigm*, Proc. of NM'98 workshop Extended version submitted for publication, 1998.

[9] I. Niemelä, P. Simons, and T. Syrjanen, *Smodels: a system for answer set programming*, Proc. of 5th ILPS Conference (2000), 1070–1080.

[10] D. Poole, A. Mackworth, and R. Goebel, *Computational intelligence - a logical approach*, Oxford University Press, Inc., 1998.

[11] P. Simons, *Smodels*, http://www.tcs.hut.fi/Software/smodels/, 1998.