Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica
Materia di Tesi: Informatica teorica

Tecnologie per l'interrogazione di basi documentarie in formato XML

Tesi di Laurea di:
Lorenzo Natile

Relatore:
Chiar.mo Prof. Andrea Asperti

II Sessione
Anno Accademico 2001/2002

Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica
Materia di Tesi: Informatica teorica

Tecnologie per l'interrogazione di basi documentarie in formato XML

Tesi di Laurea di:
Lorenzo Natile

Relatore:
Chiar.mo Prof. Andrea Asperti

II Sessione
Anno Accademico 2001/2002

A Diana
e alla mia famiglia

Indice

1 Introduzione **1**
1.1 Il progetto HELM 2
1.1.1 Requisiti di HELM 4
1.1.2 Architettura generale di HELM 5
1.1.3 Lavori correlati 6
1.2 Organizzazione della tesi 11

2 Lo standard RDF **13**
2.1 I Modelli 13
2.1.1 Formato XML 16
Reificazione 17
Formato N-triple 19
2.2 Gli Schemi 19
Le classi e le proprietà 20
I vincoli 20
2.3 Metadati in Helm 21
2.3.1 Nuovo fomato dei metadati 23

3 Un linguaggio di query per documenti XML **27**
3.1 Scenari di utilizzo 28
3.2 Modello dei dati 28
3.2.1 Tipi di nodo 29
3.2.2 Tipi di dato 29
3.3 Espressioni 29
3.3.1 Path expression 30
3.3.2 FLWR expression 31

3.3.3 Sort Expression 34
3.3.4 Espressioni condizionali 35
3.3.5 Quantified expression 35
3.3.6 Costruttori 36
Costruttori di elementi 36
Computed constructor 37
3.4 Il prologo di una query 39

4 Mathql versione alpha 41
4.1 Background matematico per la semantica del linguaggio 41
4.1.1 Costruttori di tipi 41
Tipi di dato e risultato di una query 43
4.2 Sintassi testuale e semantica di MathQL 44
4.2.1 Costrutti che denotano liste di attributed_uri 45
4.2.2 Costrutti che denotano espressioni booleane 49
4.2.3 Costrutti che denotano stringhe 51
4.3 Implementazione di un interprete 52
Architettura di Galax 53
Alcuni dettagli implementativi 54
4.3.1 Prestazioni 56

5 Linguaggio Mathql 59
5.1 Tipi di dato 60
5.1.1 Tipi primitivi 60
5.1.2 Costruttori di tipi 60
5.2 Sintassi testuale e semantica di MathQL 62
Espressioni che denotano insiemi di risorse 64
Espressioni che denotano una condizione booleana 69
Espressioni che denotano stringhe multiple 71
5.3 Sintassi testuale e semantica dei risultati delle query 73
5.4 Implementazione e sperimentazione nel progetto Helm 74
5.4.1 Interprete basato su PostgreSQL DBMS 75
Organizzazione della base di dati e dettagli implementativi 76
5.4.2 Interprete basato su Galax 79
5.5 Sperimentazione dell'interprete sui metadati della libreria di Helm 82

6 Conclusioni e sviluppi futuri **85**

A Strutture dati dell'interprete basato su Postgres **87**

B Strutture dati e funzioni XQuery per l'interprete **91**

C Esempio di interrogazione della libreria Helm **97**

Bibliografia **102**

Elenco delle figure

1.1 Il progetto HELM 3
1.2 Architettura del progetto HELM 5

2.1 Rappresentazione di uno statement 15
2.2 Uno statement e il relativo schema 21

C.1 Inserimento pattern 98
C.2 Selezione vincoli 98
C.3 Risultati dell'interrogazione 100
C.4 Visualizzazione di un oggetto 101

Elenco delle tabelle

5.1 Tabella Unicode 63
5.2 Esempio di relazione per un modello scpecifico: tabella`t4086` . . 76
5.3 Alcune tuple della relazione `registry` 77
5.4 Alcune tuple della relazione `refRel` 78
5.5 Alcune tuple della relazione `refSort` 79

Capitolo 1

Introduzione

Lo svilupppo delle tecnologie telematiche, sia per i sistemi di elaborazione che per le reti di calcolatori, ha portato ad una presenza sempre maggiore di sorgenti informative determinando una vera e propria esplosione nella quantità e varietà di dati accessibili. Poter gestire in modo efficace questa mole di dati è diventato, perciò, un problema di primaria importanza nel settore dell'IT, anche perchè l'aumento nell'offerta di informazione fatica a tradursi in un effettivo vantaggio per l'utente: questa crescita irregolare ha portato ad avere una grande varietà di sorgenti disomogenee e quindi difficilmente integrabili.

Il problema di base è l'eterogeneità dei sistemi, che può presentarsi a diversi livelli a partire dalle piattaforme hardware e software su cui una sorgente è basata (ad esempio diversi sistemi di DBMS e linguaggi di interrogazione) fino ad arrivare ai modelli dei dati (relazionale, object-oriented, ...) e agli schemi utilizzati per rappresentare la struttura logica dei dati rappresentati. In una situazione di questo tipo, risulta evidente che, per poter reperire le informazioni desiderate, sarebbe necessario avere familiarità con il contenuto, le strutture e i linguaggi di interrogazione delle singole sorgenti ed avere la possibilità di automatizzare completamente le elaborazioni.

Contestualmente alla rapida espansione e diffusione su larga scala del World Wide Web, registrata negli ultimi anni, si è assistito alla nascita di tecnologie e strumenti progettati per realizzare un'evoluzione del Web conosciuta con il nome di *Semantic Web*: un'estensione del Web in cui il contenuto informativo è "comprensibile", oltre che dagli esseri umani, anche da meccanismi automatici che possano effettuare delle scelte opportune .

L'idea è quella di realizzare un framework in cui agenti software "intelligenti" siano in grado di eseguire compiti sofisticati per conto dell'utente. Ad esempio un agente che arriva al sito di una clinica ospedaliera deve essere in grado non solo di leggere le keyword trattamento, medicina, terapia, ... ma anche di capire che il Dr. Jones lavora nella clinica il lunedì, mercoledì e venerdì. Il web semantico, nell'intento dei suoi promotori, deve garantire due livelli di interoperabilità:

- **Sintattica:** capacità di leggere i dati e ottenere una rappresentazione utilizzabile da un'altra applicazione.
- **Semantica:** capacità di comprendere il contenuto informativo dei dati.

Il primo dei due livelli, l'interoperabilità sintattica, è già da tempo raggiunta grazie all' affermarsi dell'Extendible Markup Language (XML) che permette di definire la struttura dei propri documenti ma non dice nulla sul significato delle informazioni contenute nel documento.

La convergenza verso l'interoperabilità semantica, invece, ha ricevuto un notevole impulso con l'introduzione, da parte del World Wide Web Consortium (W3C[1]), dello standard RDF (Resource Description Framework): un meccanismo *generale* di rappresentazione della conoscenza che permette di evitare qualsiasi tipo di ipotesi sul dominio applicativo nel quale avviene lo scambio, come anche la definizione (a priori) della semantica del dominio applicativo stesso: consente, cioè, di descrivere informazioni di qualsiasi natura.

1.1 Il progetto HELM

HELM[2] (Hypertextual Electronic Library of Mathematics) è un progetto a lungo termine sviluppato presso l'Università di Bologna dal Prof. Andrea Asperti e dal suo gruppo di ricerca[3]. HELM sarà presto integrato insieme ad altri progetti correlati, nella struttura del European FET Project IST-2001-33562, chiamato MOWGLY (Math On The Web: Get it by Logic and Interfaces).

[1]www.w3.org

[2]http://www.cs.unibo.it/helm

[3]I membri attuali sono: Prof. Andrea Asperti, Dott. Ferruccio Guidi, Dott. Luca Padovani, Dott. Claudio Sacerdoti Coen e Dott. Irene Schena.

Lo scopo, è la progettazione e l'implementazione di strumenti per lo sviluppo e lo sfruttamento di larghe librerie ipertestuali e distribuite di conoscenza matematica formalizzata, che comprendano tutto il materiale già codificato nei sistemi attuali. La principale novità tecnica (vedi figura 1.1) del progetto è

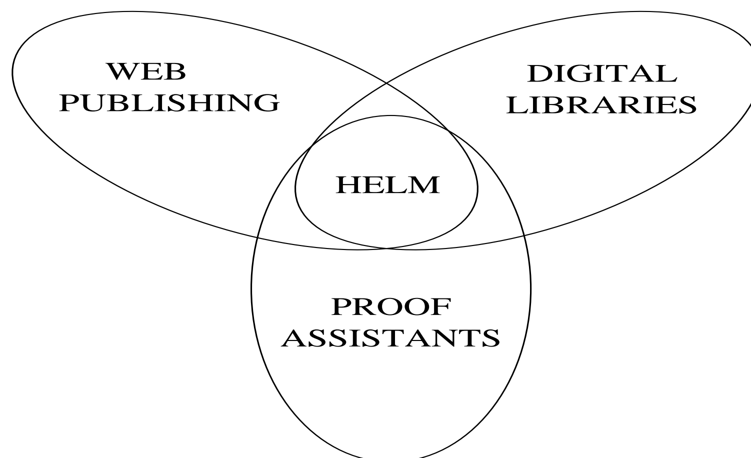

Figura 1.1: Il progetto HELM

la sinergía di diverse comunità scientifiche e argomenti di ricerca quali Digital Libraries [Com98], Web Publishing e Logical Enviroments. Dal punto di vista della pubblicazione nel Web il progetto è il primo tentativo di fornire una descrizione comprensiva di fatti matematici al fine di incrementare l'accessibilità, lo scambio e l'elaborazione attraverso il Word Wide Web. Per ottenere questi obiettivi HELM utilizza molte delle tecnologie introdotte recentemente dal W3C come XML, DOM, XSLT, XLink, Namespaces, MathML, RDF, XQuery, ...

Dal punto di vista della libreria digitale, il progetto è spinto verso uno sfruttamento di tutte le potenziali funzionalità offerte dal Web, in particolar modo un uso più integrato di browsing e facilità di ricerca. La libreria non è vista come una collezione strutturata di testi matematici, ma come una struttura virtuale che ha come scopo quello di facilitare la navigazione e la visualizzazione di informazioni a diversi livelli di dettagli. Per aumentare l'efficienza delle consultazioni, HELM ha bisogno di un buon modello di metadati per le informazioni.

Infine, l'aspetto principale è l'integrazione con i correnti tool di automazione dei ragionamenti formali (Proof Assistants). Questa integrazione facilita l'estrazione automatica dalle librerie esistenti in questi sistemi e fornisce anche

un modo per il proof-cheking di frammenti della libreria di HELM. Quindi per il successo del progetto è necessario che la comunità dei proof-assistant contribuisca attivamente alla creazione della libreria e dei tool di gestione. In particolare, è auspicabile che i moduli di esportazione dai sitemi esistenti siano realizzati dai rispettivi autori.

1.1.1 Requisiti di HELM

I principali requisiti del progetto sono:

1. **Utilizzo di formati standard.** Tutti i documenti che formano la libreria e i metadati su di essi devono essere descritti in XML. Per le elaborazioni sui documenti devono essere utilizzati, quando possibile, standard già definiti (per esempio, XSLT per le trasformazioni fra file XML).

2. **Distribuzione e replicazione.** La libreria deve essere distribuita e, per motivi di efficienza e fault-tolerance, deve prevedere l'esistenza di più copie di uno stesso documento. Per motivi di rilocabilità e bilanciamento del carico, devono essere utilizzati nomi logici per individuare gli elementi della libreria. Conseguentemente, devono essere definiti e realizzati opportuni meccanismi per la risoluzione dei nomi.

3. **Facilità di pubblicazione e accesso.** La disponibilità di uno spazio HTTP o FTP deve essere l'unico requisito per poter contribuire alla libreria. In particolare, non si può fare nessuna assunzione sul server utilizzato per la pubblicazione dei documenti, come la possibilità di mandare in esecuzione particolari programmi. Analogamente, per poter consultare la libreria, dovrà essere fornita un'interfaccia Web usufruibile con un comune browser. Interfacce di altro tipo potranno essere fornite per elaborazioni che richiedano una maggiore interattività, come la scrittura di nuovi teoremi.

4. **Modularità.** Gli strumenti utilizzati per accedere e contribuire alla libreria dovranno essere il più modulari possibile. In particolare, il livello dell'interazione con l'utente deve essere chiaramente separato da quello dell'elaborazione per permettere l'implementazione di diversi tipi di interfacce (interfacce testuali, grafiche e Web).

5. **Sfruttamento dell'informazione già codificata.** Devono essere implementati strumenti che permettano di esportare verso la libreria HELM l'informazione già codificata nei proof-assistant esistenti. Il progetto non ha scopi fondazionali: non si propone un unico formalismo logico in cui codificare tutta la libreria. Al contrario, i formalismi esistenti non devono essere eliminati o modificati, ma standardizzati descrivendoli in formato XML.

1.1.2 Architettura generale di HELM

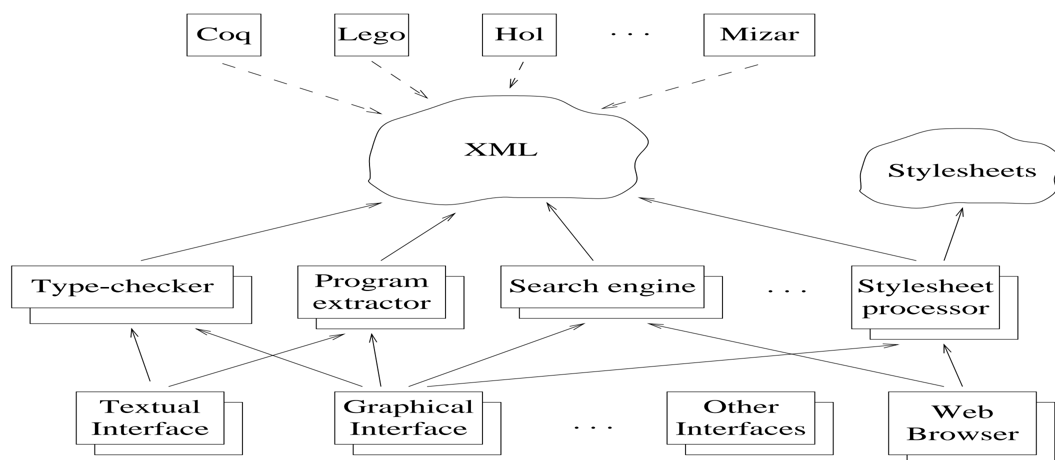

Figura 1.2: Architettura del progetto HELM

In figura 1.2 viene brevemente schematizzata l'architettura generale di HELM. La nuvola al centro, etichettata con XML, rappresenta la libreria distribuita di documenti matematici. In alto sono elencati alcuni degli attuali proof-assistant: le linee tratteggiate simboleggiano i meccanismi di esportazione da questi sistemi verso i formati standard usati da HELM. Il primo livello di moduli in basso comprende tool in grado di lavorare sul formato XML, come, per esempio, motori di ricerca. Fra questi, alcuni reimplementano in maniera modulare le funzionalità usualmente presenti nei proof-assistant, come il type-checking o la type-inference.

Una classe importante di tool è quella responsabile della trasformazione dei documenti in un formato di resa appropriato: è durante questa trasformazione che deve avvenire il processo di rimatematizzazione del contenuto, basato sulla

scelta, da parte dell'utente, di un'opportuna notazione. In conformità al primo requisito di pagina 4, l'utente può specificare le notazioni da utilizzare sotto forma di fogli di stile XSLT[4]. Processori XSLT standard possono quindi essere utilizzati per applicare le trasformazioni. I fogli di stile di notazioni matematiche formano una seconda libreria, anch'essa distribuita, rappresentata in figura dalla nuvola etichettata con XSLT.

L'ultimo livello in basso è quello delle interfacce, che verranno costruite assemblando uno o più moduli del livello soprastante, fornendone un accesso uniforme.

1.1.3 Lavori correlati

Il progetto HELM si pone nel punto di intersezione fra gli studi su digital-library, web-publishing e proof-reasoning, presentando numerosi elementi di novità rispetto allo stato dell'arte in ognuno di questi campi. Nel campo delle digital-library, il progetto più affine è EULER[5]. EULER è un progetto co-finanziato dalla Commissione Europea nel settore Telematics for Libraries il cui sviluppo è iniziato nel 1998. L'obiettivo di EULER è fornire un servizio per la ricerca di risorse informative sulla matematica quali libri, pre-print, pagine Web, abstract e spogli di articoli e riviste, periodici, rapporti tecnici e tesi. Attraverso l'interfaccia di ricerca di EULER è possibile interrogare simultaneamente un insieme di basi di dati. In particolare, il progetto si propone di integrare i seguenti tipi di risorse documentarie:

1. Basi di dati bibliografiche

2. Cataloghi di biblioteche (OPAC)

3. Riviste elettroniche prodotte da editori accademici

4. Server di pre-pubblicazioni e letteratura grigia

5. Indici di risorse matematiche in rete

[4]XSLT è lo standard per la descrizione di trasformazioni fra documenti XML. Un foglio di stile è un documento XSLT che definisce un insieme di regole di trasformazione.

[5]http://www.emis.de/projects/EULER/

Queste risorse sono integrate e messe a disposizione per mezzo di un'interfaccia utente WWW comune, il *motore di ricerca EULER*, e l'uso del protocollo Z39.50 per l'interrogazione simultanea delle diverse basi dati che condividono lo stesso schema basato sullo standard di metadati Dublin Core. La prima differenza rispetto al nostro progetto è che EULER si occupa di tutte le risorse matematiche esistenti, mentre HELM considera esclusivamente i documenti di matematica formale. Inoltre, poiché i documenti indicizzati da EULER sono tipicamente in formati orientati alla rappresentazione, come PostScript o PDF, le modalità di ricerca possibili saranno decisamente inferiori a quelle previste da HELM. È ovviamente possibile e ausplicabile l'aggiunta della libreria di HELM alla lista delle risorse catalogate.

Nell'ambito del web-publishing esistono già due standard per la rappresentazione di documenti matematici. Il primo, MathML, è uno standard del World Wide Web Consortium definito per codificare in XML sia il contenuto che la presentazione di espressioni matematiche. Il suo dominio di applicazione si estende dai sistemi per la computer-algebra al publishing, sia Web che cartaceo. Mentre la parte di MathML che si occupa della presentazione può essere efficacemente utilizzata da HELM, quella che codifica il contenuto non è direttamente fruibile per la memorizzazione di matematica formale per almeno due motivi. Il primo è la natura infinitaria della conoscenza matematica, che non può essere colta da alcun linguaggio mancante della possibilità di definire nuovi concetti. Infatti, MathML si pone l'obiettivo di definire un elemento di markup per quasi ogni operazione o entità matematica che venga studiata nelle scuole americane fino ai primi due anni di college inclusi, il che corrisponde alla conoscenza di uno studente europeo che abbia conseguito un livello A. Per tutte le altre nozioni matematiche, l'unica possibilità è quella di utilizzare un elemento generico, associandogli uno specifico markup di presentazione ed il URI di un documento, la cui forma non viene specificata, che ne descriva la semantica. Ciò è altamente insufficiente per la codifica di matematica formale, anche se fornisce un interessante linguaggio semi-formale utilizzato proficuamente in HELM durante la fase di rimatematizzazione. La precedente osservazione non si applica alla parte di MathML per la descrizione della presentazione delle espressioni in quanto questa è facilmente codificabile attraverso un numero finito di elementi di markup, come dimostra facilmente il linguaggio di LaTeX. Il secondo motivo per cui la parte di MathML per la descrizione del contenuto non è adatta alla codifica del-

la matematica formale, è che ad ogni entità matematica viene associata la sua semantica standard, che non è definita univocamente in un contesto formale. Per esempio, all'uguaglianza di MathML corrispondono nei sistemi formali diverse uguaglianze, come quelle di Leibniz a livello dei termini e dei tipi o quelle estensionale e intensionale per le funzioni. Durante la fase di rimatematizzazione, comunque, è naturale identificarle tutte con l'uguaglianza semi-formale di MathML, a patto che sia possibile risalire alle loro definizioni formali.

Il secondo standard per la codifica del contenuto matematico nell'ambito del web-publishing è OpenMath, che si preoccupa di complementare lo standard MathML definendo una semantica semi-formale per gli elementi contenutistici. L'uso consigliato di OpenMath in ambito MathML è, infatti, quello di fornire il formato dei documenti referenziati da MathML per specificare la semantica per gli elementi contenustici introdotti dagli utenti. Se si è interessati esclusivamente al contenuto, OpenMath può essere utilizzato anche separatamente da MathML. Infatti, l'obiettivo dichiarato di OpenMath è "la rappresentazione di oggetti matematici con la loro semantica, permettendone lo scambio fra programmi, la memorizzazione in database e la pubblicazione sul WWW". Concretamente, OpenMath si propone di definire un'architettura per lo scambio di espressioni matematiche basata su tre componenti: i frasari (phrasebook), i dizionari di contenuto (content dictionary, CD) e un formato XML di rappresentazione per questi e per le espressioni matematiche. I *frasari* sono interfacce software per la codifica di espressioni matematiche dal formato MathML/OpenMath a quello interno delle applicazioni orientate al contenuto, quali tool per la computer algebra come Mathematica, Maple o Derive. I *content dictionary* sono grandi tabelle, ipoteticamente una per ogni teoria matematica, che associano ad ogni entità (funzione, relazione, insieme) una sua descrizione informale e una "formale", costituita dalla sua signature. Per esempio, la descrizione informale della funzione quoziente (quotient), è "INTEGER DIVISION OPERATOR. THAT IS, FOR INTEGERS a AND b, QUOTIENT DENOTES q SUCH THAT $a = bq + r$, WITH $|r|$ LESS THAN $|b|$ AND ar POSITIVE."; la sua signature è il tipo $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ dove $\mathbb{Z}$ è l'insieme dei numeri interi. Questo tipo di semantica, che possiamo definire al più semi-formale, è sufficiente per lo scambio di informazione matematica in ambiti dove non si è interessati alle prove, ma esclusivamente alle espressioni; gli obiettivi di HELM sono però molto più ambiziosi e richiedono una semantica formale sia per i tipi che per i corpi delle definizioni. Pertanto, HELM non può

beneficiare della semantica di OpenMath. Nonostante questo, è sicuramente utile, ma non ancora implementato, l'utilizzo di OpenMath nella rappresentazione semi-formale in MathML content utilizzata nel progetto HELM.

Nell'ambito dei proof-assistant, esistono due linee di ricerca che hanno affinità con il progetto HELM. La prima è quella delle interfacce grafiche. In questo ambito, l'obiettivo è lo sviluppo di sistemi in ambiente grafico che aiutino l'utente nella visualizzazione e nella ricerca delle dimostrazioni. Si pone una grossa enfasi sul *proof-by-clicking*, ovvero sulla possibilità di applicare tattiche semplicemente selezionando il termine su cui agire e, per esempio, selezionando il lemma da applicare. Fra l'altro, il proof-by-clicking solleva l'utente dall'onere di memorizzare la sintassi, spesso complicata, del linguaggio delle tattiche dei vari proof-assistant, creando sistemi utilizzabili per interfacciarsi con proof-assistant differenti. Fra questi sistemi ricordiamo Proof General[6], che è sviluppato sotto Emacs e lavora già con diversi proof-assistant fra cui Coq e Lego, e PCoq[7] che è un'interfaccia grafica per Coq scritta in Java per motivi di portabilità e finalizzata al proof-by-clicking. Poiché, allo stadio attuale, il progetto HELM non si occupa della ricerca delle dimostrazioni, il confronto può essere effettuato solo nel campo della visualizzazione dei termini e delle prove, dove HELM si propone di raggiungere risultati sicuramente più raffinati.

La seconda linea di ricerca è quella della standardizzazione, a livello tecnologico o fondazionale, dei proof-assistant. In questo ambito sono stati proposti due grossi progetti, QED e MathWeb. QED[8], iniziato nel 1994 con la pubblicazione del QED Manifesto [Har94], persegue, come HELM, il fine della creazione di una grande libreria di documenti matematici formalizzati. Per realizzarla, viene proposto un approccio fondazionale mirato alla codifica del contenuto matematico in una logica, detta *root logic*, sufficientemente espressiva da rappresentare tutta la conoscenza matematica e, al tempo stesso, prossima alla logica "informale" delle dimostrazioni cartacee. Il progetto, a cui contribuiscono in maniera determinante gli autori di Mizar, dopo un primo slancio iniziale[9] sembra non aver conseguito grossi successi. Il motivo principale è da cercarsi nella difficoltà dell'approccio fondazionale che, sebbene interessante, non permette, almeno per

[6]http://zermelo.dcs.ed.ac.uk/ proofgen

[7]http://www-sop.inria.fr/lemme/pcoq/index.html

[8]http://www-unix.mcs.anl.gov/qed

[9]Una prima conferenza fu tenuta nel 1994 ad Argonne, seguita da una seconda a Varsavia nel 1995.

ora, il recupero dell'informazione già codificata in proof-assistant basati su logiche non banali. In mancanza di questo, la proposta si riduce al semplice sviluppo di un nuovo proof-assistant, con in più la complicazione di trovare un largo accordo, sicuramente utopistico, sulla logica da utilizzare come root logic. HELM evita accuratamente di commettere lo stesso errore[10], proponendo prima di tutto una standardizzazione a livello tecnologico, comunque propedeutica a qualsiasi studio fondazionale.

Omega è un grosso progetto intrapreso nel 1994 all'Università di Saarland dal gruppo di ricerca del Prof. Jörg Siekmann per affrontare il delicato tema dell'integrazione fra proof-assistant e theorem-prover. Successivamente, le finalità del progetto sono state estese, portando alla nascita del progetto MathWeb[11]; oggi, MathWeb[12] è sicuramente il progetto più simile ad HELM, prevedendo la creazione di una grande libreria distribuita basata su tecnologia XML.

Di seguito sono elencati documenti che forniscono una descrizione del progetto HELM e di alcuni suoi aspetti specifici:

Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata [Sac99] è una presentazione esaustiva del progetto HELM.

Studio e progettazione di un modello RDF per biblioteche matematiche elettroniche [Ric99] è uno studio preliminare, risalente al 1999, sulla possibilità di codificare i metadati della libreria di HELM nel formato standard RDF. Nonostante l'architettura di HELM descritta in questo lavoro sia stata pesantemente modificata, la parte di lavoro inerente ai metadati rappresenta un ottimo punto di partenza per le future scelte progettuali.

Sperimentazione e sviluppo di strumenti per la gestione di metadati [Lor02]descrive una serie di strumenti per la gestione di metadati con particolare riferimento alle problematiche incontrate nel definire meccanismi efficienti per l'interrogazione della libreria elettronica di Helm.

[10]Vedi il primo requisito del paragrafo **??**.

[11]http://www.mathweb.org

[12]Con Omega viene ancora indicato il sotto-progetto mirato al raggiungimento degli obiettivi originari.

Content Centric Logical Environments [APSSa] è una breve introduzione ai benefici che HELM porterà nell'ambito dei proof-assistant.

Towards a Library of Formal Mathematics [APSSc] presenta l'intero progetto nell'ottica del precedente articolo.

XML, Stylesheets and the Re-mathematization of Formal Content [APSSd] affronta il problema della rimatematizzazione del contenuto formale.

Formal Mathematics in MathML [APSSb] è una descrizione di HELM dal punto di vista del web-publishing, incentrato sull'utilizzo di MathML.

1.2 Organizzazione della tesi

Il presente lavoro di tesi è nato dall'idea di fornire soluzioni generali per l'interrogazione della base documentaria di Helm, e in particolare sperimentare tecnologie standard per la gestione delle query sui metadati relativi agli oggetti della libreria elettronica di tale progetto. Nel momento in cui il lavoro è partito era già in fase di sviluppo un sistema di Data Base Management System (DBMS) in cui esportare i metadati della libreria elettronica rappresentati in formato XML. Contestualmente al database relazionale era in fase di realizzazione un prototipo di linguaggio di interrogazione per la base documentaria ad alto livello ed una sua implementazione al livello di database.

Per via della semplice struttura dello schema RDF dei metadati e date le dimensioni della libreria elettronica, si è pensato di tentare un approccio alternativo al problema della gestione dei metadati, basato sull'uso di un linguaggio standard di interrogazione per documenti XML: XQuery.

Nel capitolo 2 viene fornita una breve introduzione ad RDF e RDF Schema, gli standard proposti dal W3C per la definizione e la descrizione di metadati.

Il capitolo 3 è interamente dedicato alla presentazione di XQuery.

Nel capitolo 4 viene descritto una versione sperimentale di linguaggio di interrogazione per la libreria dei metadati presentando la semantica operazionale e un'implementazione basata su XQuery. Vengono riportati, inoltre, i risultati dei test delle prestazioni confrontando i tempi di esecuzione delle query con quelli ottenuti dall'approccio basato sul DBMS.

Il capitolo 5 presenta e discute la semantica operazionale del linguaggio di interrogazione per metadati RDF, Mathql, di cui viene fornita un'implementazione basata su un sistema di database relazionale e se ne sperimentano le effettive potenzialità con degli esempi di interrogazioni.

Capitolo 2

Lo standard RDF

Come già detto, la naturale trasformazione del WWW in una versione evoluta, cioè dotata di una sovrastruttura di significato degli oggetti che lo compongono, è stata notevolmente alimentata dallo sforzo dei promotori del Semantic Web. In particolare il progetto Semantic Web è rivolto alla creazione di strumenti per esprimere dati e ragionamenti sui dati, che possano essere usati dalle macchine oltre che dall'uomo. Nasce cosi' l'esigenza di introdurre i metadati (letteralmente, dati sui dati) ossia informazioni che descrivono altre informazioni.

Lo standard proposto dal W3C come strumento generale per la gestione dei metadati è il Resource Description Framework [W3Ce].

RDF nasce come estensione della Platform for Internet Content Selection [W3Cd] un meccanismo per descrivere il contenuto delle pagine Web in termini di rating, cioè come indice di gradimento da parte dell'utente. RDF amplia le potenzialità di PICS: permette di descrivere risorse che non sono pagine web e consente di dare una migliore struttura stabilendo una gerarchia tra gli elementi che vengono descritti.

2.1 I Modelli

Il concetto alla base di RDF è quello di definire una risorsa come qualsiasi cosa (concreta o astratta) immaginabile, per cui è necassario un meccanismo di identificazione abbastanza potente da poter referenziare in maniera univoca sia oggetti reali che entità astratte.

I dati RDF sono dei modelli composti da nodi(corrispondenti alle risorse), attributi (proprieta' delle risorse) e valori di attributi.

Il meccanismo per descrivere le risorse è lo *statement* la cui struttura è molto simile a quella di una frase in linguaggio naturale: un statement è composto da un soggetto (risorse da descrivere), un predicato (proprietà o attributo delle risorse) e un oggetto (valore delle proprietà). Un concetto di fondamentale importanza per RDF è la possibilità di identificare qualsiasi oggetto tramite un URI, in questo modo ciascuno dei tre elementi di uno statement può essere considerato una risorsa ed eventualmente essere descritto da altri statement; per comodità il valore di un oggetto, infatti, può essere di tipo semplice come definito dallo standard XML [W3Ca]. In tal modo si può decidere il livello di dettaglio della descrizione dei dati: scegliendo che il valore di un oggetto sia una risorsa, ad esempio, si possono esprimere ulteriori informazioni su di esso (ad es. l'autore di un libro può essere descritto oltre che dal nome anche dalla nazionalità, indirizzo email, ...); assegnando un valore di tipo semplice (ad es. stringa) se non si vuole dettagliare ulteriormente l'asserzione.

In generale nel modello di dati di RDF si possono distinguere cinque tipi di oggetti:

- **Risorse**: oggetti identificati da URI come ad esempio pagine Web, parti di esse, intere collezioni di pagine e anche oggetti che non sono direttamente accessibili dal Web. Per creare identificatori unici per le risorse dichiarate RDF usa il meccanismo dei namespace XML [W3Cc].

- **Proprietà**: attributi, caratteristiche specifiche o relazioni usate per descrivere una risorsa. Ogni proprietà ha un significato specifico, definisce i valori consentiti, i tipi di risorse che può descrivere e le sue relazioni con altre proprietà.

- **Letterali**: stringhe o tipi di dato primitivi definiti da XML, usati come valori delle proprietà delle risorse.

- **Asserzioni** (statement): descrizioni di risorse in cui l'oggetto può essere un'altra risorsa o un letterale.

- **Contenitori**: collezioni di risorse o letterali. Esistono tre tipi di contenitori: bag (insiemi con ripetizioni in cui l'ordine non è rilevante), sequenze

(liste ordinate di valori eventualmente duplicati) e alternative (liste di alternative per il valore singolo di una proprietà). Tali collezioni sono molto utili in casi in cui si voglia fare riferimento a insiemi di risorse, ad esempio per descrivere informazioni su un documento creato da più autori oppure su un autore che ha scritto più di un documento.

RDF è un modello di descrizione astratto e non pone vincoli sulla sintassi e sul significato delle descrizioni di una risorsa. Ognuno, infatti, potrebbe proporre un qualsiasi meccanismo per descrivere risorse utilizzando le astrazioni previste da RDF. Ad esempio uno statement potrebbe essere rappresentato con un grafo orientato con archi e nodi eventualmente etichettati in cui le risorse sono dei nodi con archi uscenti, gli oggetti sono nodi con soli archi entranti (per una maggiore chiarezza visiva si potrebbe pensare di rappresentarli con un rettangolo) e le proprietà sono gli archi che mettono in relazione le risorse con i valori di attributi corrispondenti. Lo statement mostrato in fig. 2.1 afferma che l'autore della risorsa `"http://www.myhost.org/ mbianchi"` è Mario Bianchi che ha e-mail `"mbianchi@myhost.org"`.

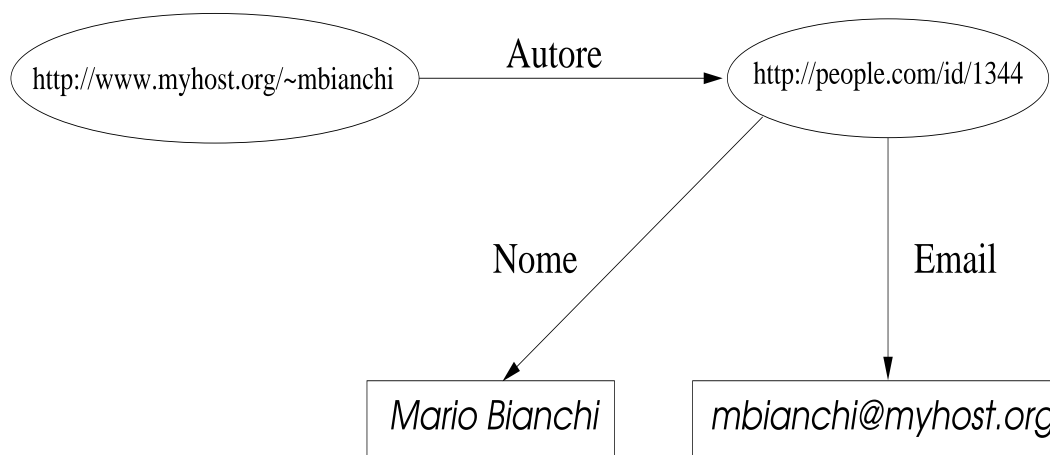

Figura 2.1: Rappresentazione di uno statement

L'organizzazione fisica degli statement RDF segue due formati distinti: la descrizione formulata dal W3C, infatti, propone la possibilità di usare una sintassi basata su XML e un formato più semplice definito *N-triple*(triple notation,[W3Cb]).

La sintassi che prevede l'uso di XML come metalinguaggio di descrizione

assume due forme: una estesa, detta *seralization syntax*, ed una abbreviata introdotta per facilitare la scrittura manuale di documenti RDF.

Con il crescente uso di XML e la possibilità di avere a disposizione una serie di strumenti già collaudati per questo formato (parser, validatori, ...) si ha un maggiore uso del primo dei due formati, sebbene il formato N-triple sia altrettanto espressivo e permetta di costruire descrizioni del tutto analoghe a quelle realizzabili con XML.

2.1.1 Formato XML

Un documento RDF contenente uno statement generalmente ha come tag di apertura e chiusura il tag `RDF` che solitamente è associato al prefisso di namespace `rdf`. Con la stessa semantica usata nei documenti XML, infatti, i documenti RDF fanno largo uso del meccanismo dei namespace anche per identificare gli schemi [W3Cf] per l'interpretazione degli statement.

Il tag usato in uno statement per descrivere risorse è `rdf:Description` che può essere accompagnato dall'attributo `rdf:about` nel caso in cui si voglia fare asserzioni su risorsre già esistenti. Per creare nuove risorse il tag `rdf:about` viene omesso e si può usare l'attributo `rdf:ID` per assegnare un identificatore alla nuova risorsa. Ad esempio il seguente documento RDF contiene tre esempi di statement:

```
<rdf:RDF
  xmlns:rdf="http://w3c.org/99/02/22-rdf-syntax-ns#"
  xmlns:s:"http://authordescription/schema#">

  <rdf:Description rdf:about="http://www.myhost.org/~mbianchi">
    <s:Autore>Mario Bianchi</s:Autore>
  </rdf:Description>


  <rdf:Description rdf:about="http://www.myhost.org/~mbianchi">
    <s:Autore rdf:resource="http://people.com/id/1285"/>
  </rdf:Description>
```

```
  <rdf:Description rdf:about="http://people.com/id/1285">
    <s:Nome>Mario Bianchi</s:Nome>
    <s:Email>bianchi@myhost.org</s:Email>
  </rdf:Description>}

<rdf:RDF>
```

Il primo statement descrive la risorsa `http://www.myhost.org/ mbianchi` affermando che l'autore di tale risorsa è `Mario Bianchi`.

Nel secondo statement la proprietà `Autore` della stessa risorsa è rappresentata a sua volta da una risorsa che viene descritta in modo più dettagliato dal terzo statement (tale esempio è rappresentato graficamente in fig.2.1). Si noti l'uso del prefisso di namespace `s:` usato per indicare lo schema esterno in cui sono definite tutte le proprietà.

La sintassi usata nell'esempio precedente è detta serialization syntax e presenta le metainformazioni nel modo più chiaro possibile. Esistono, tuttavia, delle regole che consentono di abbreviare le descrizioni degli statement RDF. Ad esempio il terzo statement del modello precedente può essere abbreviato in:

```
<rdf:Description rdf:about="http://people.com/id/1285"
  s:Nome="Mario Bianchi"
  s:Email="bianchi@myhost.org" />
```

La regola usata è quella relativa alle proprietà che hanno come valore un letterale infatti in tali casi, se la proprietà non è ripetuta più volte all'interno dello statement, può essere trasformata in un attributo dell'elemento Description.

Reificazione

(auto-referenzialità) Con questo meccanismo è possibile definire descrizioni che riguardano altri statement, detti statement di ordine superiore, in altre parole si possono fornire meta-informazioni su meta-informazioni: occorre considerare la meta-informazione come una risorsa da descrivere. Ad esempio per esprimere la frase: *Lucio afferma che Mario Bianchi è l'autore della risorsa "http://www.myhost.org/ mbianchi"* si deve attribuire la proprietà `afferma` allo statement `Mario Bianchi è l'autore della risorsa "http://www.myhost.org/ mbianchi"`. Ad esempio sono equivalenti:

```
<rdf:Description rdf:about="http://www.myhost.org/~mbianchi">
  <s:Autore>Mario Bianchi</s:Autore>
</rdf:Description>

<rdf:Description>
  <rdf:subject rdf:resource="http://www.myhost.org/~mbianchi"/>
  <rdf:predicate rdf:resource="/myschema.rdf#Autore"/>
  <rdf:object>Mario Bianchi</rdf:object>
  <rdf:type
   rdf:resource="rdf:Statement"/>
</rdf:Description>
```

Come si può notare nell'esempio le proprietà necessarie per descrivere uno statement di ordine superiore sono:

type : indica il tipo di descrizione che si vuole effettuare, per descrivere uno statement il valore deve essere: `"rdf:Statement"`; può essere usata anche in contesti diversi dalla reificazione, ad esempio per specificare il tipo di contenitore a cui una certa risorsa appartiene.

subject : usata per specificare la risorsa da descrivere.

predicate : l'attributo utilizzato per descrivere il soggetto.

object : il valore dell'attributo

Per descrivere uno statement reificato possiamo scrivere:

```
<rdf:Description>
  <rdf:subject rdf:resource="http://www.myhost.org/~mbianchi"/>
  <rdf:predicate rdf:resource="/myschema.rdf#Autore"/>
  <rdf:object>Mario Bianchi</rdf:object>
  <rdf:type rdf:resource="rdf:Statement"/>
  <s:AffermatoDa>Lucio</s:AffermatoDa>
</rdf:Description>
```

Formato N-triple

Consiste in una rappresentazione testuale degli elementi del modello, in cui il primo elemento della tripla è la risorsa da descrivere, il secondo è la proprietà e infine si ha il vaolre della proprietà. Gli esempi visti in sintassi XML diventano:

```
<http://www.myhost.org/~mbianchi, Autore, Mario Bianchi>
<http://www.myhost.org/~mbianchi, Autore, http://people.com/id/85>
<http://people.com/id/1285, Nome, Mario Bianchi>
<http://people.com/id/1285, Email, bianchi@myhost.org>
```

2.2 Gli Schemi

Il modello di dati di RDF non fornisce strumenti per dichiarare proprietà e per definire relazioni tra queste proprietà e altre risorse.

Lo standard proposto dal W3C per assolvere a queste funzioni è *RDF Schema*. A differenza di XML Schema o del Document Type Definition XML, RDF Schema non pone dei vincoli sulla struttura del documento ma fornisce informazioni utili all'interpretazione degli statement definiti nei modelli RDF.

Mentre con l'uso del Document Type Definition e di XML Schema si possono imporre dei vincoli sintattici sulla struttura dei documenti (ad esempio indicando che i possibili valori di un attributo devono essere dei caratteri PC-DATA), gli schemi RDF costituiscono uno strumento di validazione semantica permettendo di controllare il significato assunto dalle proprietà e dalle risorse utilizzate negli statement.

Oltre a fornire un meccanismo di base, definito in termini di RDF stesso, per un sistema di tipizzazione da utilizzare nei modelli RDF, RDF Schema definisce un insieme di risorse RDF da usare per descrivere caratteristiche di altre risorse e proprietà RDF.

Per evitare confusione tra definizioni indipendenti, possibilmente incompatibili, degli stessi elementi di un modello vengono usati i namespace XML.

In termini pratici gli schemi RDF vengono introdotti per creare un `vocabolario` di proprietà specifiche del contesto in cui vengono fatte le descrizioni. Si possono, inoltre, specificare classi di oggetti a cui associare determinate proprietà con meccanismi molto simili a quelli usati nei linguaggi `object-oriented`. La caratteristica che differenzia in modo sostanziale gli schemi RDF dai linguaggi

ad oggetti è la possibilità di descrivere nuove proprietà (assimilabili ai metodi di una classe) senza includerle in una classe di risorse: mentre nei linguaggi object oriented l'unico modo per definire nuovi tipi è l'ereditarietà tra classi (si definiscono classi come insiemi di attributi), negli schemi RDF si definiscono proprietà come relazioni fra classi indicandone i possibili valori e il dominio di applicazione. L'enfasi maggiore è riservata alle proprietà e non alle classi.

Le classi e le proprietà

Il sistema di tipi di RDF Schema contiene un insieme predefinito di proprietà e risorse che costituiscono le unità base utilizzate per la creazione di nuovi elementi. Le classi previste dalle specifiche formali sono:

- `rdfs:Resource`: la classe per la rappresentazione delle risorse, ogni risorsa è istanza di questa classe.
- `rdfs:Literal`: è una sottoclasse di rdfs:Resource e rappresenta un letterale, consente di utilizzare valori corrispondenti ai tipi semplici di XML.
- `rdf:Property`: rappresenta le proprietà ed è sottoclasse di rdfs:Resource
- `rdfs:Class`: permette di specificare un insieme di risorse con caratteristiche comuni, ogni classe di risorse è di questo tipo.

Per creare nuovi elementi si usa la proprietà `rdf:type` che esplicita direttamente il tipo della risorsa, in questo caso la costruzione dell'elemento avviene per istanziazione. Nella costruzione per ereditarietà, invece, le proprietà a disposizione sono: `SubClassOf` per definire nuove classi e `SubPropertyOf` per la creazione di nuove proprietà. In fig. 2.2 viene mostrato lo statement di fig. 2.1 esplicitandone lo schema.

I vincoli

Una caratteristica fondamentale di RDF è rappresentata dalla possibilità di esprimere vincoli sull'utilizzo di risorse e proprietà. I predicati più utilizzati per esprimere vincoli su altre proprietà sono:

- `rdfs:range` (codominio) Se usato come proprietà di una risorsa, indica le classi che saranno oggetto di un'asserzione che ha tale risorsa come predicato.

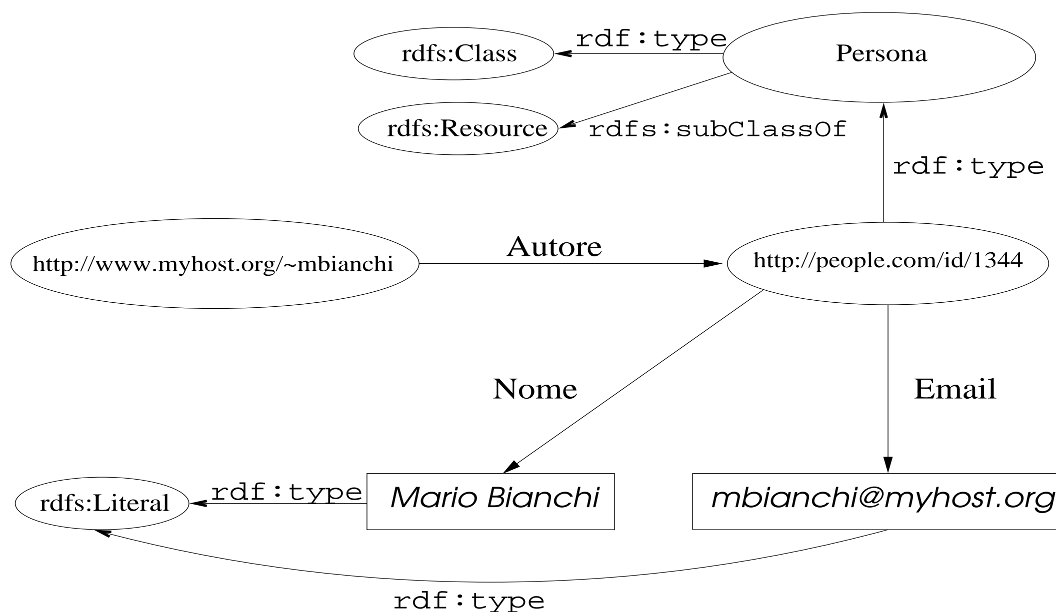

Figura 2.2: Uno statement e il relativo schema

- **rdfs:domain** (dominio) Usato come attributo di una risorsa, indica le classi(soggetto) a cui la risorsa può essere applicata.

2.3 Metadati in Helm

Le metainformazioni relative agli oggetti della libreria elettronica del progetto Helm si possono dividere in due categorie:

Metadati *intrinseci*, ricavati dalla rappresentazione strutturata degli oggetti della libreria per descrivere le dipendenze logiche esistenti fra i diversi elementi (teoremi e definizioni) che compongono un documento;

Metadati *estrinseci*, nel senso che non possono essere estratti direttamente dai dati, per esprimere informazioni ausiliarie come ad esempio l'autore di un documento, il titolo e la descrizione.

L'organizzazione dei metadati segue una gerarchia di classi di risorse in cui i metadati estrinseci, detti anche Dublin Core e compiutamente descritti in [dub], corrispondono ad un insieme di atributi assegnabili a singoli documenti o a intere teorie.

Le metainformazioni sulle dipendenze logiche tra gli oggetti della libreria sono descritte usando opportune proprietà RDF che modellano la sementica di dipendenza logica attraverso due tipi di relazioni: dipendenza *forward*, per descrivere quali oggetti sono necessari alla costruzione di una dimostrazione che porta alla definizione di un nuovo termine; dipendenza *backward*, per specificare la dipendenza inversa ossia per esprimere quali sono gli elementi di una teoria che dipendono dall'oggetto che si vuole descrivere.

Al momento dell'avvio del presente lavoro di tesi lo schema RDF dei metadati era molto semplice e, di conseguenza, il contenuto informativo dei file XML contenenti gli statement relativi alle descrizioni degli oggetti della libreria era piuttosto limitato. Le principali classi che componevano lo schema erano:

- `MathResource`: la classe radice, sottoclasse di rdf:Resource, è usata per rappresentare qualsiasi risorsa di tipo matematico descritta all'interno della libreria.

- `Object`: usata per modellare un singolo termine (definizione o teorema) di una teoria, sottoclasse di MathResource.

- `DirectoryOfObject`: un raggruppamento di singoli oggetti per formare teorie.

- `Occurrence`: usata per specificare correttamente il concetto di dipendenza e di contesto in cui un termine viene utilizzato (nell'ipotesi, nella tesi,...).

Le proprietà definite nello schema per esprimere le dipendenze logiche erano `refObj` per le relazioni forward e `backPointer` per le relazioni backward; per specificare le metainformazioni inerenti al contesto di utilizzazione di un termine si usavano le proprietà `occurrence` e `position`; per i metadati Dublin Core, infine, erano state definite le corrispondenti proprietà (`author`, `title`, ...).

I valori assunti dagli attributi elencati erano di tipo stringa, ad eccezione della proprietà `position` che poteva assumere valori in un insieme di stringhe prestabilito in cui ogni valore corrispondeva ad uno specifico contesto in cui un termine può comparire:

- `MainHypothesis`: ipotesi principale del teorema;

- `InHypothesis`: ipotesi del teorema;

- `MainConclusion`: tesi principale del teorema;
- `InConclusion`: tesi del teorema;
- `InBody`: corpo della dimostrazione di un teorema.

Il seguente frammento di file XML mostra le metainformazioni relative all'oggetto `cic:/Coq/Init/Datatypes/sum.ind#xpointer(1/1/2)`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
 xml:lang="en"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:h="http://www.cs.unibo.it/helm/schemas/mattone.rdf#">
  <h:Object
   rdf:about="cic:/Coq/Init/Datatypes/sum.ind#xpointer(1/1/2)">
    <h:refObj>
      <h:Occurrence>
        <h:position>MainConclusion</h:position>
        <h:occurrence>
          cic:/Coq/Init/Datatypes/sum.ind#xpointer(1/1)
        </h:occurrence>
      </h:Occurrence>
    </h:refObj>
  </h:Object>
</rdf:RDF>
```

2.3.1 Nuovo fomato dei metadati

In seguito, per arricchire le descrizioni degli oggetti della libreria in modo da aumentare il contenuto informativo dei relativi metadati, abbiamo ristrutturato lo schema RDF[1] aggiungendo nuove classi e proprietà.

In particolare è stata introdotta la classe `Position` che ha come istanze le classi: `MainHypothesis, InHypothesis, MainConclusion, InConclusion, InBody`.

[1] http://www.cs.unibo.it/helm/schemas/schema-helm#

Per catturare le informazioni relative agli elementi `<Rel>` (usati nei documenti XML degli oggetti[2] della libreria per rappresentare una variabile dichiarata precedentemente) è stata introdotta la proprietà `refRel`, mentre per esprimere metainformazioni sugli elementi `<Sort>` si è deciso di usare la poprietà `refSort`.

Per ragioni pratiche e implementative si è deciso di mantenere i metadati relativi agli elementi `<Rel>` e `<Sort>` di tutti gli oggetti della libreria del progetto Helm in due singoli file XML: `forward_rel.xml` contenente gli statement relativi agli elementi `<Rel>` e `forward_sort.xml` per memorizzare le descrizioni degli elementi `<Sort>`.

La nuova proprietà `depth` è utilizzata sia negli statement che descrivono gli oggetti sia nelle descrizioni delle metainformazioni relative agli elementi `<Rel>` e `<Sort>`.

Negli statement relativi agli oggetti `depth` compare solo nei riferimenti (sia forward che backward) in cui la property `position` assume il valore MainConclusion o MainHypothesis e rappresenta l'indice di De Brujin usato per indicare il numero di nodi che bisogna risalire nel backbone della dimostrazione per arrivare all'inizio (o meglio il numero di applicazioni che si incontrano risalendo all'indietro il backbone della dimostrazione).

Nei metadati che descrivono i `<Rel>` e i `<Sort>`, invece, l'attributo `depth` indica il numero di occorrenze che separano le variabili, rappresentate da tali elementi, dalla relativa dichiarazione. La proprietà `sort` è usata, esclusivamente nei metadati degli elementi `<Sort>`, per specificare il tipo dell'elemento `<Sort>`, cioè il valore assunto dall'attributo `value` di tale elemento, nei documenti XML che rappresentano gli oggetti della libreria. Di seguito viene riportato lo stesso esempio di metadati della sezione precedente espresso nel nuovo formato:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE rdf:RDF [
  <!ENTITY rdfns 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <!ENTITY hthns 'http://www.cs.unibo.it/helm/schemas/schema-helmth#'>
  <!ENTITY hns 'http://www.cs.unibo.it/helm/schemas/schema-helm#'>
   ]>
<rdf:RDF xml:lang="en" xmlns:rdf="&rdfns;"
```

[2] La DTD è disponibile all'URL: http://www.cs.unibo.it/ sacerdot/tesi/tesi009.html

```
         xmlns:h="&hns;" xmlns:hth="&hthns;">
  <h:Object
   rdf:about="cic:/Coq/Init/Datatypes/sum.ind#xpointer(1/1/2)">
    <h:refObj rdf:parseType="Resource">
      <h:position rdf:resource="&hns;MainConclusion"/>
      <h:depth>3</h:depth>
      <h:occurrence>
        <h:Object
 rdf:about="cic:/Coq/Init/Datatypes/sum.ind#xpointer(1/1)"/>
      </h:occurrence>
    </h:refObj>
  </h:Object>
</rdf:RDF>
```

Capitolo 3

Un linguaggio di query per documenti XML

Con il crescente impiego di XML per la memorizzazione, per lo scambio e per la presentazione di informazione acquista maggiore rilevanza l'abilità di interrogare in modo intelligente sorgenti di informazione che adottano tale standard. Per sfruttare uno dei più grossi vantaggi di XML, cioè la sua flessibilità nel rappresentare diversi tipi di informazione da sorgenti di dati differenti, il linguaggio di query deve essere tale da consentire ricerche e interpretazioni di dati da diverse fonti di informazione.
XQuery è un linguaggio funzionale progettato per essere di dimensioni ridotte, semplice da implementare e tale da realizzare query concise e di facile interpretazione. La sua flessibilità consente di interrogare un ampio spettro di sorgenti di informazione inclusi database e documenti.
La intassi di XQuery è "human-readable", le specifiche per una sintassi basata su XML sono descritte in [W3Cp]. XQuery deriva da un liguaggio di query per XML chiamato Quilt che a sua volta eredita caratteristiche da vari linguaggi: XPath [W3Ch], XQL [W3Cj], XML-QL[Ali], SQL, e OQL.
La versione 1.0 di XQuery contiene XPath Version 2.0 come sottoinsieme e qualsiasi espressione che è sintatticamente valida e viene eseguita con successo sia da XQery che da XPath restituisce lo stesso risultato in entrambi i linguaggi.
XQuery è un linguaggio fortemente tipato in cui gli operandi delle espressioni, gli operatori e le funzioni devono essere conformi ad un tipo ben specificato. Le regole per assegnare i tipi alle espressioni sono descritte in un apposito docu-

mento del W3C [W3Cn]. Le definizioni dei tipi possono anche essere importate da un XML Schema. L'esecuzione di una query comprende una fase di tipaggio statico e una fase di tipaggio dinamico. Il valore generato dalla valutazione dinamica è garantito essere un'istanza del tipo assegnato dal tipaggio statico, proprietà di "soundness". Il controllo statico dei tipi può essere disabilitato dall'utente e una query che passa il type checking almeno una volta, ritornerà sempre lo stesso risultato anche se il controllo dei tipi non viene effettuato.

3.1 Scenari di utilizzo

Gli scenari in cui è utile un linguaggio d'interrogazione per documenti XML sono molteplici:

- Eseguire query su singoli o collezioni di documenti per ritrovare singoli documenti, tabelle dei contenuti, per generare nuovi documenti come risultati di una query.
- Eseguire query "document-oriented" e "data-oriented" su documenti che contengono dati quali cartelle cliniche, cataloghi di prodotti, ...
- Eseguire query su file di configurazione, files di log rappresentati in XML.
- Eseguire query su strutture DOM che ritornano insiemi di nodi.
- Eseguire interrogazioni sulla rappresentazione XML di dati contenuti in una generica sorgente di dati per estrarre informazioni, per rappresentare le informazioni estratte in formato XML, o per integrare informazioni provenienti da sorgenti eterogenee.

3.2 Modello dei dati

Un modello dei dati è l'insieme dei concetti usati per descrivere un insieme di dati, le loro associazioni, e le operazioni che agiscono sui dati stessi. Dato che i documenti XML hanno una struttura ad albero, il modello dei dati di XQuery(come illustrato in dettaglio in [W3Cm]) è basato sul concetto di albero con nodi etichettati e con il concetto di identità del nodo per gestire i riferimenti XML (IDREF, ...).

3.2.1 Tipi di nodo

Il modello dei dati si basa sul concetto di *nodo*. In XQuery abbiamo sette tipi di nodo:

- document;
- element;
- value;
- attribute;
- namespace (NS);
- processing insttruction (PI);
- comment;

Per ogni tipo di nodo è definito un costruttore e l'effetto del costruttore è quello di creare un nuovo nodo con identità diversa da qualsiasi altro nodo. Oltre ai nodi, il modello dei dati supporta due tipi di collezioni: le liste (ordinate) e gli insiemi (non ordinati), le collezioni non possono essere innestate.

3.2.2 Tipi di dato

I nodi contengono valori appartenenti al dominio di un tipo di dato definito da XML-Schema.

```
SchemaType ::= SimpleSchemaType | ComplexSchemaType
```

Un `simple type` può essere un tipo primitivo (string, boolean, float, double, ID, IDREF) o derivato. Un complex type definisce il contenuto e la struttura di un elemento.

3.3 Espressioni

L'unita base di XQuery è l'espressione. Le varie tipologie di espressioni offerte dal linguaggio, i cui operandi sono altre espressioni, si costruiscono a partire da parole chiave, simboli e operandi. Per elaborazioni più complesse, inoltre, si

possono usare operatori e funzioni built-in di XQuery definite in [W3Co] che si basano sui tipi di dato specificati in . Le espressioni primitive di base del linguaggio sono dette *primary expression* e includono letterali, variabili, chiamate di funzioni e l'uso di parentesi per controllare la precedenza degli operatori. Per esprimere le condizioni booleane si possono usare le *logical espression* che producono sempre un valore booleano `true` o `false`, mentre per i valori numerici XQuery fornisce gli operatori per addizione, sottrazione, divisione, moltiplicazione e modulo nella forma unaria e binaria. Altri tipi di espressioni sono: path expressions, FLWR expressions, costruttori di elementi XML, espressioni condizionali, espressioni di ordinamento, espressioni quantificate. Il valore di un'espressione è sempre una sequenza. Una sequenza è una collezione ordinata di zero o più item. Un *item* è un valore atomico (Atomic value) o un nodo. Un *Atomic value* è un valore contenuto nello spazio di valori di un XML Schema Atomic type descritti in [W3Ck]. . Il contenuto di *nodo* ha tipo conforme a quelli descritti in [W3Cm] tra cui valori tipati (typed values), stringhe e nomi(name). Il *typed value* di un nodo è una sequenza di zero o più valori semplici. Un nodo *stringa* è un'istanza del tipo `xs:string` (dove il prefisso di namespace xs: indica il namespace di XML Schema http://www.w3.org/2001/XMLSchema). Il *name* di un nodo è un'istanza del tipo `xs:QName`.

3.3.1 Path expression

La sintassi delle path expression di XQuery può essere considerata un sottoinsieme esteso di XPath, infatti pur essendo molto simile ad XPath (e definita in termini di XPath) presenta alcune divergenze. La differenza più evidente è che le espressioni XQuery restituiscono sequenze ordinate di nodi mentre il valore di una espressione XPath è un insieme non ordinato di nodi. Nel caso di XPath, infatti, per effettuare operazioni che richiedono la conoscenza di informazioni relative all'ordine in cui disposti gli elementi del documento c'è bisogno di consultare ulteriormente il documento originario. Una path expression può essere usata per localizzare nodi all'interno di un albero, generalmente è composta da una serie di uno o più `step` separati da "/" e che possono cominciare con "/" o "//". Uno step è composto da due parti principali: un espressione che genera una sequenza di nodi e un predicato usato per filtrare i nodi in tale sequenza.

La prima parte di uno step è a sua volta formata da un (`axis`) che definisce la direzione del movimento (*child, descendant, parent, ...*) e da un `Node Test` che specifica il tipo e/o il nome dei nodi che lo step deve selezionare (*QName, text, comment, ...*). Un predicato consiste in un espressione, chiamata `predicate expression` e racchiusa tra parentesi quadrate, usata per selezionare nodi in una sequenza. Alcuni esempi di step che contengono predicati sono i seguenti:

- Per selezionare il secondo elemento `"chapter"` che è figlio del nodo contesto:

  ```
  child::chapter[2]
  ```

- Per selezionare tutti i discendenti del nodo contesto il cui nome è `"toy"` e il cui attributo `"color"` ha valore `"red"`:

  ```
  descendant::toy[attribute::color = "red"]
  ```

- Per selezionare tutti gli `"employee"` figli del nodo contesto che hanno un sottoelemento `"secretary"`:

  ```
  child::employee[secretary]
  ```

Come si può notare dagli esempi nella sintassi non abbreviata un axis è separato da un Node Test da "::" mentre nella sintassi abbreviata l'axis `child::` può essere omesso (è l'axis di default) ed altri possono essere sostituiti da simboli (ad es. la forma abbreviata di `attribute::` è @). Gli esempi di sopra usando la sintassi abbreviata si riscrivono:

- `chapter[2]`
- `descendant::toy[@color = "red"]`
- `employee[secretary]`

3.3.2 FLWR expression

Per iterare la valutazione di espressioni e per legare variabili a risultati intermedi si usano le FLWR expressions. Questo tipo di espressioni viene anche usato per

eseguire join tra due o più documenti e per ristrutturare dati. Il nome "FLWR" deriva dalle parole chiave `for, let, where, return` che indicano le quattro clausole che caratterizzano le espressioni FLWR, tali clausole sono interpretate come segue:

- La clausola `for` associa una o più variabili alle espressioni. In particolare vengono create tuple di legami di variabili (nel senso di sequenze ordinate) estratte dal prodotto Cartesiano delle sequenze di valori restituite dalle espressioni che seguono tale clausola.

- Una clausola `let` lega una o più variabili al risultato della valutazione dell'espressione. Se sono presenti clausole `for` i legami di variabili creati dalla `let` vengono aggiunti alle tuple generate dalle clausole `for`, altrimenti le clausole `let` generano una tupla con tutti i legami di variabili.

- La clausola `where` viene usata per selezionare elementi dalle tuple di legami di variabili generate dalle clausole `for` e `let`. L'espressione nella clausola `where` viene valutata una volta per ognuna delle suddette tuple e se il valore è `true` i legami di variabili nella tupla vengono usati nella clausola `return`.

- Con la clausola `return` si costruisce il risultato di tutta l'espressione FLWR. La `return-expression` viene valutata una volta per ognuna delle tuple restituite dalle clausole `let` e `for`, dopo aver eliminato quelle che non soddisfano l'eventuale clausola `return`. Il risultato della FLWR expression consiste in una sequenza ordinata formata dai valori restituiti dall'espressione.

Il modo di legare le variabili delle clausole `for` e `let` è differente. Nella `let` ogni variabile è legata direttamente al valore dell'espressione considerato nella sua interezza, ad esempio nella seguente query:

```
let $s := (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

la variabile `$s` è legata al valore dell'espressione `(<one/>, <two/>, <three/>)` e poiché non ci sono clausole `for` la clausola `let` genera una tupla che contiene il valore legato a `$s`. L'output sarà:

```
<out>
    <one/>
    <two/>
    <three/>
</out>
```

Una query simile ma con una clausola `for`:

```
for $s in (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

restituisce:

```
<out>
    <one/>
</out>
<out>
    <two/>
</out>
<out>
    <three/>
</out>
```

In questo esempio la variabile `$s` è legata tre volte: la prima volta a `<one/>`, la seconda a `<two/>` e la terza a `<three/>`; ciò accade perché quando si ha un unica espressione nella clausola for il prodotto Cartesiano equivale alla sequenza di valori generata dall'espressione. Per ogni valore di tale sequenza viene generata una tupla e per ognuna di esse viene invocata la clausola `return`. Nella query che segue, invece, le tuple vengono generate dal prodotto Cartesiano dei valori delle espressioni nella clausola `for`:

```
for $i in (1, 2),
    $j in (3, 4)
return
    <tuple>
        <i>{ $i }</i>
        <j>{ $j }</j>
    </tuple>
```

Il risultato è:

```
<tuple>
    <i>1</i>
    <j>3</j>
</tuple>
<tuple>
    <i>1</i>
    <j>4</j>
</tuple>
<tuple>
    <i>2</i>
    <j>3</j>
</tuple>
<tuple
    <i>2</i>
    <j>4</j>
</tuple>
```

3.3.3 Sort Expression

Per ordinare i valori generati dalle espressioni si può usare la parola chiave `sort by`. Il valore dell'espressione a sinistra di tale parola chiave è detta *input sequence* mentre i suoi elementi sono chiamati *input item*. Le espressioni a destra di `sort by` sono dette *ordering expression* e vengono valutate una volta per ogni input item (*inner focus*). Ogni input item, in questo modo, viene associato al valore della relativa ordering esspression e l'ordinamento degli input item viene effettuato considerando tali associazioni. Se sono presenti più ordering expression il principale criterio di ordinamento è quello della ordering expression più a sinistra, le altre vengono considerate procedendo da sinistra verso destra. Ogni ordering expression può essere seguita dalle keyword `ascending` o `descending` per specificare la direzione dell'ordinamento, se nessuna delle due keyword è presente la clausola di default è `ascending`. Altre parole chiave da utilizzare dopo una ordering expression sono `empty greatest` e `empty least`: si possono usare per indicare la posizione, nella sequenza di output, degli input item per cui la ordering expression restituisce la sequenza vuota. La seguente query, ad

esempio, elenca tutti i libri con prezzo maggiore di 100 ordinandoli per primo autore e nei gruppi di libri con lo stesso primo autore i libri sono ordinati per titolo:

```
//book[price > 100] sort by (author[1], title])
```

Per indicare che l'ordine del risultato della valutazione di un'espressione non è rilevante si può anteporre all'espressione la parola chiave `unordered`. In questo modo si possono ottimizzare le prestazioni dell'implementazione che può generare il risultato nel modo più efficiente possibile.

3.3.4 Espressioni condizionali

Vengono espresse in XQuery usando le parole chiave `if`, `then` e `else`. Se il valore booleano della *test-expression* è `true` viene valutata l'espressione che segue la keyword `then` altrimenti viene valutata l'espressione nel ramo `else`. Ad esempio nella query che segue la text-expression controlla se esiste un attributo chiamato `discounted` indipendentemente dal suo valore:

```
if ($part/@discounted)
    then $part/wholesale
    else $part/retail
```

3.3.5 Quantified expression

Usate per esprimere la quantificazione universale ed esistenziale queste espressioni restituiscono sempre un valore booleano. Una quantified expression è formata da un quantificatore, una o più *in-clause*, dalla keyword `satisfies` e da un test-expression. In particolare il quantificatore è rappresentato dalla keyword `some` o `every` mentre la clausola `in` serve per legare una variabile al valore di un'espressione: come avviene nella clausola `for` della FLWR expression, le in-cause generano tuple di legami di variabili usando valori estratti dal prodotto Cartesiano delle espressioni coinvolte. La text-expression è l'espressione principale da cui dipende il valore dell'intera quantified expression e viene valutata per ogni tupla generata dalle clausole `in`. Il valore di output della quantified expression viene determinato secondo le regole:

1. Quando si usa il quantificatore `some`, la quantified expression è `true` se tra i valori restituiti dalla test-expression almeno uno è `true`, altrimenti la quntified expression è `false`. Nei casi in cui la clausola `in` non genera alcuna tupla di legami di variabili il risultato è `false`.

2. Se viene usata la keyword `every`, il valore della quantified expression è `true` se ogni valore restituito dalla text-expression è `true` altrimenti il risultato è `false`. Quando la clausola `in` non restituisce alcuna tupla il risultato della quantified expression è `true`.

Nell'esempio che segue, il risultato è `true` se ogni elemento `part` ha un attributo `discount`:

```
every $part in //part satisfies $part/@discounted
```

La seguente query, invece, restituisce `true` se almeno un elemento `employee` soddisfa la condizione nella text-expression:

```
some $emp in //employee satisfies ($emp/bonus > 0.25 * $emp/salary)
```

3.3.6 Costruttori

All'interno di una query è possibile creare strtrutture XML usando i costruttori. XQuery supporta costruttori elementi, attributi, processing instruction, sezioni CDATA e commenti (per gli ultimi tre tipi di nodi la sintassi usata è quella XML standard). Un particolare tipo di costruttore chiamato *computed constructor* è usato per creare elementi o attributi, con nomi ottenuti da altri elementi e attributi, e per costruire interi documenti.

Costruttori di elementi

Sono usati per costruire elementi e attributi (anche attributi per dichiarare namespace) XML con nome e contenuto costante (cioè non ottenuto da una qualche computazione). La notazione usata è quella standard di XML e il nome usato nel tag di chiusura deve essere uguale a quello usato nel tag di apertura. Il seguente codice crea un elemento `libro` che contiene attributi, sottoelementi e testo:

```
<book isbn="isbn-0060229357">
    <title>Harold and the Purple Crayon</title>
    <author>
        <first>Crockett</fisrst>
        <last>Johnson</last>
    </author>
</book>
```

All'interno di un costruttore per distinguere un espressione dal testo vero e proprio si usano le parentesi graffe , tali espressioni sono dette *enclosed expression*. La seguente query mostra l'uso delle enclosed expression:

```
<example>
    <p> Ecco una query </p>
    <eg> $i//title </eg>
    <p> Ecco il risultato della query di sopra </p>
    <eg>{ $i//title }</eg>
</example>
```

Il risultato sarà:

```
<example>
    <p> Ecco una query </p>
    <eg> $i//title </eg>
    <p> Ecco il risultato della query di sopra </p>
    <eg><title>Harold and The purple Crayon</title></eg>
</example>
```

Una enclosed expression può essere usata anche per ottenere il valore di un attributo:

```
<book isbn="{$i/@booknum}" />
```

Computed constructor

Un computed constructor inizia con la keyword `element`, `attribute` o `document`. Per creare un elemento o un attributo le relative parole chiave sono seguite dal nome del nodo da creare mentre i documenti non hanno nome. Il nome può

essere specificato con un QName(come descritto in [W3Cl] o ottenuto con una enclosed expression (che ritorna un QName) ed è seguito da una enclosed expression che genera il contenuto del nodo. La query che segue produce lo stesso output del primo esempio della sezione precedente:

```
element book
{
    attribute isbn { "isbn-0060229357" },
    element author
    {
        element first { "Crockett"},
        element last { "Johnson"}
     }
}
```

Come già anticipato una caratteristica importante dei computed constructor è che consentono di generare il nome di un nodo a partire da nomi di altri nodi. L'esempio che segue mostra come il nome di un elemento può essere tradotto da una lingua ad un'altra, la variabile `$dict` è associata a una sequenza di elementi che compongoino un dizionario simili a:

```
<entry word="address">
    <variant lang="German">Addresse</variant>
    <variant lang="Italian">indirizzo</variant>
</entry>
```

Supponendo che la variabile `$e` sia legata al seguente elemento:

```
<address>123 Roosvelt Ave. Fludhing, NY 11368</address>
```

Allora per tradurre il nome dell'elemento legato a `$e` in Italiano e lasciare invariato il contenuto di tale elemento si può scrivere:

```
element
    {xf:expanded-Qname("",
        xf:data($dict/entry[word=name($e)]/variant[lang="italian"]))}
    {$e/node()}
```

dove la prima enclosed expression genera il nome dell'elemento mentre la seconda genera il contenuto. Il risultato è il seguente:

```
<indirizzo>123 Roosvelt Ave. Flushing, NY 11368</indirizzo>
```

Per costruire un intero documento XML si può usare la keyword `document` seguita da una enclosed expression che contenga esattamente un nodo elemento al top-level che funga da elemento radice del documento.

```
document
{
    <author-list>
        document("bib.xml")//book/author
    </author>
}
```

3.4 Il prologo di una query

Una query può essere divisa in due blocchi principali: la parte iniziale detta *query prolog* contenente una serie di dichiarazioni che costituiscono l'ambiente operativo in cui la query viene processata e il *query body* costituito da una sequenza di espressioni che definiscono il risultato della query. Nel prologo si possono inserire sia dichiarazioni di prefissi di namespace che dichiarazioni di nuovi namespace di default per elementi e funzioni oltre a quelli predefiniti di XQuery. La dichiarazione di un prefisso di namespace contiene le keyword `declare namespace`, il prefisso da introdurre e l'URI del namespace. Ad esempio la seguente dichiarazione specifica il prefisso `foo`:

```
declare namespace foo = "http://example.org"
```

Nella parte iniziale della query, inoltre, si possono dichiarare funzioni definite dall'utente: tale dichiarazione specifica il nome della funzione, i nomi e i tipi dei parametri il tipo del risultato e il *body function* che definisce il modo in cui il risultato della funzione è calcolato a partire dai parametri. La dichiarazione che segue, ad esempio, introduce una funzione che a partire da un sequenza di elementi `catalogue` restituisce tutti i titoli dei libri pubblicati nell'anno 2001:

```
define function titles(element catalogue* $c)
    returns element title*
{
```

```
    for $i in $c/book[year=2001]
    return
        <title>
        $i/title
        </title>
}
```

Capitolo 4

Mathql versione alpha

In questo capitolo viene presentato un prototipo di linguaggio per l'interrogazione di metadati progettato a partire dalle esigenze specifiche della libreria elettronica del progetto Helm. Oltre alla sintassi e alla semantica operazionale che descrivono il linguaggio, viene illustrata un'implementazione basata sull'utilizzo, a livello più basso, dello standard proposto dal W3C per l'interrogazione di documenti in formato XML: XQuery. Nella parte finale vengono illustrati i risultati dei test delle prestazioni effettuati confrontando i tempi di esecuzone ottenuti dalla suddetta implementazione, con quelli ottenuti dall'interprete basato sul DBMS realizzato da Domenico Lordi[Lor02].

4.1 Background matematico per la semantica del linguaggio

4.1.1 Costruttori di tipi

Il costruttore `Setof Y` indica il tipo degli insiemi finiti (sequenze finite e non ordinate) di elementi in `Y`, senza ripetizioni. Partendo da questo costruttore possiamo definire i seguenti operatori su insiemi:

- $\emptyset$: (`Setof Y`)
- $\exists$: ((`Setof Y`) $\rightarrow$ `Boole`) $\rightarrow$ `Boole`
- $\in$: `Y` $\rightarrow$ (`Setof Y`) $\rightarrow$ `Boole` (infisso)

- $\subseteq$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ `Boole` (infisso)

- $\between$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ `Boole` (infisso)

- $\cap$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ (`Setof Y`) (infisso)

- $\cup$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ (`Setof Y`) (infisso)

- $\sqcup$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ (`Setof Y`) (infisso, unione disgiunta)

  Da notare che poiché gli insiemi sono finiti l'esistenziale è sempre decidibile, come è spiegato in[GS]. $Y * Z$ indica il prodotto dei tipi Y e Z e i suoi elementi sono le coppie (y, z) tali che $y : Y$ e $z : Z$. $Y \rightarrow Z$ indica il tipo delle funzioni da Y a Z e fy indica l'applicazione di $f : Y \rightarrow Z$ a $y : Y$. Le relazioni sui tipi, come l'uguaglianza, sono viste come funzioni a valori in `Boole`.

- `Fst`:(`Y` $\times$ `Z`) $\rightarrow$ `Y` tale che `Fst`$(y, z) = y$.

- `FstSet`:(`Setof`(`Y`$\times$`Z`)) $\rightarrow$ (`Setof Y` tale che `FstSet`$W = \{$`Fst`$w \mid w \in W\}$

- $W(y)$: (`Setof`(`Y` $\times$ `Z`)) $\rightarrow$ `Y` $\rightarrow$ `Z`, nel caso in cui W contenga una sola coppia il cui primo elemento è y. Negli altri casi $W(y)$ non è definito.

- $W[y \rightarrow z]$: (`Setof`(`Y`$\times$`Z`)) $\rightarrow$ `Y` $\leftarrow$ `Z` $\rightarrow$ (`Setof`(`Y`$\times$`Z`)) è l'insieme ottenuto da W eliminando ogni coppia il cui primo elemento è y e aggiungendo la coppia (y, z).

- $U + W$ è l'unione di due insiemi di coppie come segue:

$$U + \emptyset \qquad \text{diventa} \qquad U$$

$$U + (W \sqcup \{(y, z)\}) \qquad \text{diventa} \qquad U[y \leftarrow] + W$$

  Gli ultimi tre operatori sono usati per manipolare gli association set cioè insiemi di coppie tali che le prime componenti di due coppie distinte sono sempre diverse. Tali insiemi saranno usati per formalizzare i contesti di valutazione.

Tipi di dato e risultato di una query

Il risultato di una query MathQL è un insieme di elementi della libreria Helm (URI) che presentano determinate caratteristiche; tali caratteristiche possono riguardare il formato dell'URI, l'esistenza di dipendenze logiche, particolari valori del set di metadati Dublin Core oppure una combinazione di questi.

Per ragioni implementative è comodo mantenere una struttura dati interna per memorizzare i valori delle proprietà (in seguito si userà anche il termine attributi) associate agli URI in modo da usarle nelle varie espressioni del linguaggio per "filtrare" ulteriormente i risultati delle query.

I tipi di dato degli oggetti del linguaggio sono:

- Un insieme di attributi A è un association set che mette in relazione i nomi di attributi ai loro valori, il tipo è T_0 = `Setof(mq_svar`$\times$ `String)`. È usato per rappresentare il set di proprietà associato ad un URI.

- Una reference (URI reference) r è un oggetto di tipo `String`.

- Una property (proprietà Dublin Core) p è un oggetto di tipo `String`.

- Un'attributed_uri U associa ad una reference un insieme di attributi e una property, il tipo è: T_1 = `(String` $\times T_0 \times$ `String)`. Il valore della property, di cui al punto precedente, rappresenta un'informazione usata per ordinare i risultati delle query.

- Un insieme di attributed_uri (formato interno del risultato di una query) S ha tipo: $T_2 =$ `Setof` T_1.

  Ad esempio il risultato della query: "Trovre l'insieme degli oggetti referenziati da `cic:/Algebra/Basics/NEG_anti_convert.con`" è rappresentato internamente nel seguente modo:

```
{"cic:/Coq/ZArith/fast_integer/fast_integers/add_un.con";
"$position" = "InBody"; ""}
{"cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)";
"$position" = MainHypothesis"; ""}
{"cic:/Coq/ZArith/fast_integer/fast_integers/anti_convert.con";
"$position" = InConclusion"; ""}
{"cic:/Coq/Init/Logic/Equality/eq.ind#xpointer(1/1)";
```

```
"$position" = MainConclusion"; ""}
...
```

Si noti che poichè nella query non era richiesto nessun tipo di ordinamento, la property viene rappresentata con una stringa vuota.

- Il risultato restituito all'esterno è un insieme di stringhe V con tipo: `Setof String`.

  Il risultato della query dell'esempio precedente ha il seguente formato:

```
{"cic:/Coq/ZArith/fast_integer/fast_integers/add_un.con";
"cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)";
"cic:/Coq/ZArith/fast_integer/fast_integers/anti_convert.con";
"cic:/Coq/Init/Logic/Equality/eq.ind#xpointer(1/1)"}
...
```

4.2 Sintassi testuale e semantica di MathQL

Le espressioni del linguaggio si dividono in tre categorie:

- Espressioni che denotano liste di attributed_uri, esse appartengono alla produzione grammaticale `<mq_list>` e la loro semantica è data dalla relazione di valutazione infissa $\Downarrow_l$.
- Espressioni che denotano condizioni booleane, si ottengono con la produzione grammaticale `<mq_boole>` e la loro semantica è data dalla relazione di valutazione infissa $\Downarrow_b$.
- Le espressioni che denotano stringhe appartengono alla produzione grammaticale `<mq_string>` e la rispettiva relazione di valutazione è $\Downarrow_s$ q

Le espressioni possono contenere stringhe tra virgolette (quoted string), la cui sintassi è:

```
<hex>          ::= '0 -9' |'A - F' | 'a - f']
<escaped>      ::= "u" <hex> <hex> <hex> <hex> | '"' | "\" | "^"
<mq_string>    ::= '"' [ "\" <escaped> "^" | '^ "\^' ]* '"'
```

L'operazione di estrazione viene eseguita da `Unquote` che ha tipo: `String` → `String`. Le query possono anche contenere variabili da associare alle risorse (rvar), variabili da associare agli insiemi di risorse (svar) e variabili per le stringhe multiple (vvar).

```
<alpha>    ::= [ 'A - Z' | 'a - z']+
<number>   ::= [ '0 - 9' ]+
<id>       ::= <alpha> [ <alpha> | <number> ]*
<mq_rvar>  ::= <id>
<mq_lvar>  ::= "%" <id>
<mq_svar>  ::= "$" <id>
```

Il contesto di valutazione delle espressioni $\Gamma = (\Gamma_l, \Gamma_r)$ è una coppia di association set che legano rispettivamente le lvar e le rvar agli attributed_uri. Il tipo K di Γ è:

$$\texttt{Setof(<mq_lvar>} \times T_1\texttt{)} \times \texttt{Setof(<mq_rvar>} \times T_1\texttt{)}$$

e i tipi delle relazioni di valutazione:

$$\begin{aligned}
\Downarrow_l &: (K \times \texttt{<mq_alist>}) \rightarrow \texttt{Setof } T_1 \rightarrow \texttt{Boole} \\
\Downarrow_b &: (K \times \texttt{<mq_boole>}) \rightarrow \texttt{Boole} \rightarrow \texttt{Boole} \\
\Downarrow_s &: (K \times \texttt{<mq_string>}) \rightarrow \texttt{String} \rightarrow \texttt{Boole}
\end{aligned}$$

4.2.1 Costrutti che denotano liste di attributed_uri

```
<path_prefix>   ::= [ (^ :/#*?" ) ]+ | "*"
<path_body>     ::= <string>
<path_fragment> ::= [ "/" <number> | "/**" | "/*" ]*
<path>          ::= "" | <path_prefix> ":" <path_body>
                    [ "#1" <path_fragment> ]?
<mq_set>        ::= "reference" <string>
                  | "pattern" <path>
                  | <mq_lvar> | <mq_rvar>
                  | "(" <mq_set> ")"
                  | "use" <mq_set> "position" <mq_svar>
                  | "usedby" <mq_set> "position" <mq_svar>
```

```
                | "select" <mq_rvar> "in" <mq_set>
                  "where" <mq_boole>
                | <mq_set> "union" <mq_set>
                | <mq_set> "intersect" <mq_set>
                | "let" <mq_lvar> "be" <mq_set> "in" <mq_set>
```

Il costrutto `let` introduce le `lvar` nel contesto Γ_l, formalmente:

$$\frac{i: \texttt{<mq_lvar>} \quad ((\Gamma_l,\Gamma_r),x_1) \Downarrow_l S_1 \qquad ((\Gamma_l[i \leftarrow S_1],\Gamma_r,x_2) \Downarrow_l S_2}{((\Gamma_l,\Gamma_r),\ \text{let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_l S_2}$$

I costrutti per leggere le variabili dal contesto e per la parentesizzazione sono:

$$\frac{i: \texttt{<mq_lvar>}}{((\Gamma_l,\Gamma_r),i) \Downarrow_l \Gamma_l(i)} \qquad \frac{i: \texttt{<mq_rvar>}}{((\Gamma_l,\Gamma_r),i) \Downarrow_l \Gamma_r(i)} \qquad \frac{(\Gamma,x) \Downarrow_s S}{(\Gamma,(x)) \Downarrow_l S}$$

$\Gamma_l(i)$ e $\{\Gamma_r(i)\}$ valutano $\emptyset$ se i non è definita.

Il costrutto `union` produce l'unione di due insiemi di attributed_uri nel seguente modo:

$$\frac{(\Gamma,x_1) \Downarrow_s S_1 \quad (\Gamma,x_2) \Downarrow_s S_2}{(\Gamma,x_1 \text{ union } x_2) \Downarrow_s S_1 \oplus S_2}$$

dove due attributed_uri con la stessa URI reference (appartenenti a due insiemi diversi) vengono incluse da $\oplus$ solo se anche gli insiemi di attributi sono identici:

1 $(S_1 \sqcup \{(r,A_1,p)\}) \oplus (S_2 \sqcup \{(r,A_2,p)\})$ diventa $S_1 \oplus S_2 \oplus \{(r,A_1,p)\}$

se $A_1 = A_2$

$(S_1 \sqcup \{(r,A_1,p)\}) \oplus (S_2 \sqcup \{(r,A_2,p)\})$ diventa $S_1 \oplus S_2$

se $A_1 \neq A_2$

2 $S_1 \oplus S_2$ diventa $S_1 \cup S_2$

$\oplus$ è definito associativo e la regola 1 ha precedenza sulla regola 2

Il costrutto `intersect` genera l'intersezione di due insiemi di attributed_uri nel seguente modo:

$$\frac{(\Gamma, x_1) \Downarrow_s S_1 \quad (\Gamma, x_2) \Downarrow_s S_2}{(\Gamma, x_1 \text{ intersect } x_2) \Downarrow_s S_1 \otimes S_2}$$

per le URI reference che compaiono in entrambe le liste $\otimes$ fa l'unione delle liste di attrubuti: se le due liste hanno un attributo con lo stesso identificatore ma con valore diverso allora le due URI reference vengono escluse dall'intersezione.

1 $\quad (S_1 \sqcup \{(r, A_1, p)\}) \otimes (S_2 \sqcup \{(r, A_2, p)\}) \quad$ diventa $\quad S_1 \otimes S_2$

se $\exists (a_1, s_1) \in A_1,\ (a_2, s_2) \in A_2$ tale che $a_1 = a_2$ *ma* $s_1 \neq s_2$

$(S_1 \sqcup \{(r, A_1, p)\}) \otimes (S_2 \sqcup \{(r, A_2, p)\}) \quad$ diventa $\quad S_1 \otimes S_2 \otimes U$

altrimenti

2 $\quad S_1 \otimes S_2 \quad$ diventa $\quad S_1 \cap S_2$

dove $U = \{(r, A_1 \cup A_2, p)\}$, $\oplus$ è definito associativo, e la regola **1** ha precedenza sulla regola **2**

Il costrutto `diff` fa la "differenza" tra due insiemi di attributed_uri come segue:

$$\frac{(\Gamma, x_1) \Downarrow_s S_1 \quad (\Gamma, x_2) \Downarrow_s S_2}{(\Gamma, x_1 \text{ diff } x_2) \Downarrow_s S_1 \ominus S_2}$$

dove $X \ominus Y$ prende solo gli attributed_uri di X che non appartengono a Y; gli uri di X che appartengono anche a Y vengono inclusi nel risultato solo se possiedono almeno un attributo con lo stesso identificatore ma con valore diverso:

1 $\quad (S_1 \sqcup \{(r, A_1, p)\}) \ominus (S_2 \sqcup \{(r, A_2, p)\}) \quad$ diventa $\quad S_1 \ominus S_2 \cup \{(r, A_1, p)\}$

se $\exists (a_1, s_1) \in A_1,\ (a_2, s_2) \in A_2$ tale che $a_1 = a_2$ *ma* $s_1 \neq s_2$

$(S_1 \sqcup \{(r, A_1, p)\}) \ominus (S_2 \sqcup \{(r, A_2, p)\}) \quad$ diventa $\quad S_1 \ominus S_2$

altrimenti

2 $S_1 \ominus S_2$ diventa S_1

dove $\oplus$ è definito associativo, e la regola **1** ha precedenza sulla regola **2**

Il costrutto `ref` (reference) trasforma un insieme di reference in un insieme di attributed_uri, ognuno con un insieme di attributi vuoto (cioè trasforma il tipo `Setof String` nel tipo T_2 definito in 4.1.1):

$$\frac{V : \texttt{Setof String}}{(\Gamma,\ \text{ref } x) \Downarrow_s \{(s, \emptyset, p) \quad | \quad s \in V\}}$$

Il costrutto `pattern` denota l'insieme di attributed_uri (ciascuno con insieme di attributi nullo) le cui reference fanno match con il pattern fornito:

$$\frac{V : \texttt{Setof String}}{(\Gamma,\ \text{pattern } x) \Downarrow_s \{(r, \emptyset, p) \quad | \quad (\exists s \in V) \quad r \in \ \text{Pattern } s\}}$$

dove `Pattern` s rappresenta l'insieme di reference che fanno match con l'espressione regolare s, tra quelle disponibili nella base di dati.

Il costrutto `select` estrae da un insieme di attributed_uri quelli che soddisfano la condizione booleana data. La condizione è testata per ogni attributed_uri dell'insieme e le risorse sono assegnate a una rvar (nel contesto Γ_r prima del test.

$$\frac{i : \texttt{<rvar>} \quad (\Gamma, x_1) \Downarrow_s S}{(\Gamma,\ \text{select } i \text{ in } x_1 \text{ where } x_2) \Downarrow_s \ \texttt{Select} \ S\, i\, \Gamma\, x_2}$$

dove la funzione `Select` e' definita come segue:

$$\frac{i : \texttt{<rvar>}}{\texttt{Select} \ \emptyset\, i\, \Gamma\, x \text{ diventa } \emptyset}$$

$$\frac{i : \texttt{<rvar>} \quad ((\Gamma_l, \Gamma_r[i \leftarrow R]), x) \Downarrow_b F}{\texttt{Select}(S \sqcup \{U\})\, i\, \Gamma\, x \text{ diventa } \texttt{Select} S\, i\, \Gamma\, x}$$

$$\frac{i : \texttt{<rvar>} \quad ((\Gamma_l, \Gamma_r[i \leftarrow R]), x) \Downarrow_b T}{\texttt{Select}(S \sqcup \{U\})\; i\; \Gamma\; x \text{ diventa } (\texttt{Select} S\; i\; \Gamma\; x) \cup U}$$

dove U è una tripla della forma (r, A, p) e $\Gamma = (\Gamma_l,\, \Gamma_r)$

Il costrutto `use` restituisce l'insieme di attributed_uri che usano gli elementi dell'insieme di attributed_uri dato (dipendenze backward):

$$\frac{i : \texttt{<svar>} \qquad (\Gamma, x_1) \Downarrow_s S}{(\Gamma,\ \text{use } x \text{ position } i) \Downarrow_s \{\{(r_2, A_2, p_2)\} \mid \exists\, (r_1, A_1, p_1),\ (r_1, r_2) \in R,\ (i, s) \in A_2\}}$$

dove la relazione R specifica la dipendenza backward.

Il costrutto `usedby` restituisce l'insieme di attributed_uri che sono usati dagli elementi dell'insieme di attributed_uri dato (dipendenze forward):

$$\frac{i : \texttt{<svar>} \qquad (\Gamma, x_1) \Downarrow_s S}{(\Gamma,\ \text{use } x \text{ position } i) \Downarrow_s \{\{(r_2, A_2, p_2)\} \mid \exists\, (r_1, A_1, p_1),\ (r_1, r_2) \in R',\ (i, s) \in A_2\}}$$

dove la relazione R' specifica la dipendenza forward.

Il costrutto `sortedby` ordina una lista di attributed_uri in base ad una data proprietà p (metadati Dublin Core):

$$\frac{p : \texttt{<string>} \qquad (\Gamma, x_1) \Downarrow_s S}{(\Gamma, x \text{ sortedby } p) \Downarrow_s \{\{(r, A, p)\} \mid (r, A, p) \in S,\ (r, p) \in S_o\}}$$

in cui S_o è l'insieme delle coppie (r, p) ordinate in base al secondo elemento della coppia.

4.2.2 Costrutti che denotano espressioni booleane

```
<mq_boole> ::= "false" | "true"
             | "(" <mq_boole> ")"
             | "not" <mq_boole>
             | <mq_boole> "and" <mq_boole>
```

```
| <mq_boole> "or" <mq_boole>
| <mq_set> "subset" <mq_set>
| <mq_set> "setequal" <mq_set>
| <mq_string> "is" <mq_string>
```

Alcuni costrutti per costanti, operazioni standard e parentesizzazione:

$$\frac{}{(\Gamma, false) \Downarrow_b F} \quad \frac{}{((\Gamma, true) \Downarrow_b T}$$

$$\frac{(\Gamma, x) \Downarrow_b b}{(\Gamma, (x)) \Downarrow_b b} \quad \frac{(\Gamma, x) \Downarrow_b T}{(g, not\, x) \Downarrow_b F} \quad \frac{(\Gamma, x) \Downarrow_b F}{(g, not\, x) \Downarrow_b T}$$

$$\frac{(\Gamma, x_1) \Downarrow_b F}{(\Gamma, x_1\ and\, x_2) \Downarrow_b F} \quad \frac{(\Gamma, x_1) \Downarrow_b T \quad (\Gamma, x_2) \Downarrow_b b}{(\Gamma, x_1\ and\, x_2) \Downarrow_b b}$$

$$\frac{(\Gamma, x_1) \Downarrow_b T}{(\Gamma, x_1\ or\ x_2) \Downarrow_b T} \quad \frac{(\Gamma, x_1) \Downarrow_b F \quad (\Gamma, x_2) \Downarrow_b b}{(\Gamma, x_1\ or\ x_2) \Downarrow_b b}$$

Si nota che le operazioni binarie sono valutate con una strategia early-out.

I costrutti `subset` e `setequal` confrontano due insiemi di attributed_uri. `subset` verifica che tutti gli elementi del primo insieme siano presenti anche nel secondo. `setequal` controlla se i due insiemi dati sono uguali (in entrambi i costrutti il test riguarda solo gli uri, gli attributi vengono ignorati).

$$\frac{(\Gamma, x_1) \Downarrow_s S_1 \quad (\Gamma, x_2) \Downarrow_s S_2}{(\Gamma, x_1\ subset\, x_2) \Downarrow_b (S_1 \subseteq' S_2)} \quad \frac{(\Gamma, x_1) \Downarrow_v S_1 \quad (\Gamma, x_2) \Downarrow_v S_2}{(\Gamma, x_1\ setequal\, x_2) \Downarrow_b (S_1 =' S_2)}$$

dove $\subseteq'$ e $='$ operano solo sulla prima componente delle triple che formano gli attributed_uri.

Il costrutto `is` confronta due stringhe:

$$\frac{(\Gamma, x_1) \Downarrow_s s_1 \quad (\Gamma, x_2) \Downarrow_s s_2}{(\Gamma, x_1\ is\ x_2) \Downarrow_b (s_1 = s_2)}$$

in cui l'uguaglianza tra stringhe è case-sensitive.

4.2.3 Costrutti che denotano stringhe

Queste espressioni compaiono come operandi del costrutto `is`.

```
<rest_string> ::= "'" [ (^ ' ) ]* "'"
<mq_strconst> ::= "mainhypothesis" | "inhypothesis"
                |  "mainconclusion" | "inconclusion"
                |  "inbody"
<mq_string>   ::= <restr_string>
                |  <mq_strconst>
                |  <mq_rvar>
                |  <mq_svar>
                |  "func" <string> <mq_rvar>
```

`mainhypothesis, inhypothesis, mainconclusion,inconclusion, inbody`: stringhe costanti *MainHypothesis, InHypothesis, MainConclusion, InConclusion* e *InBody.*

Un insieme di stringhe può essere ottenuto esplicitamente con i seguenti costrutti:

$$\frac{q_1 : \texttt{<string>} \;\ldots\; q_m : \texttt{<string>}}{(\Gamma, \{q_1, ..., q_m\}) \Downarrow_v \{\; \texttt{Unquote}\; q_1, ...,\; \texttt{Unquote}\; q_m\}} \qquad \frac{q : \texttt{<quoted_string>}}{(\Gamma, q) \Downarrow_v \{\texttt{Unquote}\; q\}}$$

I costrutti per leggere le variabili dal contesto sono:

$$\frac{i :\; \texttt{<mq_rvar>}}{((\Gamma_l, \Gamma_r), i) \Downarrow_s \texttt{First}\, \Gamma_r(i)} \qquad \frac{i :\; \texttt{<mq_svar>}}{((\Gamma_l, \Gamma_r), i) \Downarrow_s \texttt{Fetch}\, \Gamma_r\; i}$$

dove `Fetch` Γ_r i ricerca la prima occorrenza della variabile i nella lista di attributi dell'ultimo elemento inserito in Γ_r; `First` $\Gamma_r(i)$ prende il primo elemento della tripla restituita da $\Gamma_r(i)$ (attributed_uri) e $\Gamma_r(i)$ valuta $\emptyset$ se i non è definita.
Si noti che il costrutto per leggere le `rvar` differisce da quello presentato nella sezione precedente poiché restituisce solo la URI reference dell'attributed_uri associato alla `rvar`, e non l'intero attributed_uri.

Il costrutto `func` restituisce il valore di una data proprietà (metadati Dublin Core) associato ad un attributed_uri:

$$\frac{f : \texttt{<string>} \quad i : \ \texttt{<mq_rvar>}}{(\Gamma, \ \ \texttt{func} f \ i) \Downarrow_s \texttt{Fun} \ \ f \ (\texttt{First}\, \Gamma_r(i))}$$

in cui `Fun` $f\, r$ restituisce il valore della proprietà f associata all'uri reference r.

4.3 Implementazione di un interprete

Al momento dell'avvio del presente lavoro di tesi era in fase di realizzazione l'implementazione del *back-end* di un interprete per il linguaggio presentato nelle sezioni precedenti a cura di Domenico Lordi[Lor02]; tale implementazione è basata sull'uso di un sistema di DBMS per gestire le ricerche relative alle interrogazioni sui metadati della libreria del progetto Helm.

La soluzione realizzata da Lordi prevede una strutturazione e un'organizzazione della base di dati e, più in generale di tutto il software implementato, fortemente legate allo schema RDF dei metadati e alla struttura interna dei file XML che li contengono.

Per sperimentare la possibilità di fornire soluzioni più generali, cioè adattabili con minimo sforzo implementativo a eventuali modifiche degli schemi RDF dei metadati si è seguito un approccio alternativo basato sull'utilizzo di uno standard del W3C: XQuery.

Come già anticipato in precedenza, XQuery non è un linguaggio di interrogazione specifico per metadati RDF. Tuttavia, data la semplicità degli schemi dei metadati della libreria del progetto Helm (vedi sez.2.3), oltre all'approccio "classico", cioè quello basato sull'uso di un sistema DBMS, si è deciso di sperimentare le potenzialità offerte dalla famiglia di linguaggi di interrogazione per documenti XML poposti come satndard dal W3C. Per testare la possibilità di eseguire le query direttamente sui file dei metadati, memorizzati in formato XML, si è scelto di usare un'implementazione di XQuery: Galax[Posa], un pacchetto applicativo sviluppato in open source da Lucent - Bell Labs[1] e AT&T

[1]http://db.bell-labs.com/

Labs Research[2] che supporta completamente le specifiche di XQuery definite nei working draft del W3C.

Una delle principali motivazioni che ha influito sulla scelta del suddetto tool, tra le diverse implementazioni disponibili di XQuery, è rappresentata dal fatto che Galax è implementato in Ocaml[OCa]: ciò oltre a facilitare l'integrazione col il resto del software sviluppato nell'ambito del progetto Helm ha fatto presuppore delle prestazioni, in termini di tempi di elaborazione, nettamente superiori a quelle delle altre implementazioni sviluppate al momento in cui ha avuto inizio il presente lavoro di tesi.

Architettura di Galax

Galax prende in input un documento XQuery: un file con estensione `.xq` contenente espressioni in sintassi XQuery come ad esempio il comando per aprire un documento XML o per importare un XML Schema. I documenti XQuery, i documenti XML e gli XML Schema vengono sottoposti al parser che che ne restituisce l'albero di sintassi astratta (AST). La rappresentazione AST viene poi "mappata" in una struttura dati chiamata *Datamodel*: è un formato molto simile all'interfaccia *DOM* ma presenta alcune differenze.

La sintassi XQuery è molto complessa, per questo motivo il W3C [W3C] ha proposto una sintassi semplificata, un sototinsieme della sintassi completa ma con lo stesso potere espressivo, detta *Core XQuery Syntax*. La semantica di valutazione delle query e il sistema dei tipi sono definite in base alla sintassi Core XQuery. Il modulo di Galax preposto alla trasformazione dalla sintassi completa alla sintassi Core è il *mapper*: prende in input l'albero di sintassi astratta costruito dal parser e ritorna il corrispondente Core XQuery AST. Le specifiche da seguire per effettuare questo "mapping" sono definite in appositi documenti del W3C [3]. Oltre alla suddetta trasformazione, il mapper viene usato anche per estrarre da un XML Schema le corrispondenti dichiarazioni di tipi.

Il type-checking è effettuato sulla rappresentazione Core XQuery AST delle query e le regole di semantica statica e per l'inferenza dei tipi sono definite nella sezione *Static Semantics* del documento *Formal Semantic* [4]. La valutazione delle query viene anch'essa eseguita sul Core XQuery AST in accordo con le

[2] http://www.research.att.com/

[3] `http://www.w3.org/TR/query-semantics/#sec_core_mapping_declarations`

[4] `http://www.w3.org/TR/query-semantics/#sec_typerules`

regole definite nella parte riguaradnte la semantica dinamica del documento Formal Semantic[5]. Oltre al parser di default Galax offre la possibilità di usare un *SAX parser* per ottimizzare l'uso della memoria principale.

Per eseguire una query si usa il file eseguibile `galax.opt` (versione ottimizzata del query engine) che prende in input un file contenente la query ed eventualmente un file contenente il contesto di valutazione (dichiarazioni di namespace, dichiarazioni di tipi, ...) e visualizza a video il risultato della query.

Per avere un riscontro immediato delle prestazioni di Galax e, poiché i comandi del linguaggio che effettuano la ricerca delle informazioni nella base documentaria sono solo un sottoinsieme dei costrutti del linguaggio Mathql, si è deciso di implementare solo questi ultimi costrutti: si sono riutilizzati, cioè, tutti i moduli che nell'implementazione legata al DBMS non interagiscono direttamente con la base di dati. Come visto in precedenza le query di Mathql sono ottenute come composizione delle espressioni che denotano i vari costrutti, di conseguenza risulta scomodo eseguire le query XQuery, generate per i costrutti di Mathql, memorizzandole in appositi file per usare il file eseguibile `galax.opt`. Per tali ragioni si è dovuto modificare il codice di alcuni moduli di Galax: in particolare per sottoporre una query al query engine si è scelto di usare il formato stringa usando una funzione del modulo `Toputils` (`Toputils.eval_querystring`). La suddetta funzione restituisce il risultato della query nel formato interno di Galax, per adattare l'output delle query al formato usato dagli altri costrutti dell'interprete si sono apportate delle modifiche ai tipi dei dati restituiti in output. Il modulo `Toputils` con le suddette modifiche è stato compilato con gli altrimoduli di Galax ed è stato esportato nel codice dell'interprete come libreria pre-compilata.

Alcuni dettagli implementativi

Nell'implementazione del comando `pattern` si è dovuto creare un file XML (`getallrdfuris.xml`) contenente gli URI di tutti i file dei metadati della libreria. Il file è stato usato per ricercare, eseguendo un "pattern-matching" tra stringhe, tutti gli URI dei file di metadati che soddisfano l'espressione regolare fornita: tale ricerca è stata implementata usando le funzioni della libreria

[5] `http://www.w3.org/TR/query-semantics/#sec_dynamic`

`Str` di Ocaml poiché la funzione built-in di XQuery `xf:match` non era ancora supportata da Galax.

I costrutti `use`, `usedby` e `func` son stati implemetati costruendo una query (con sintassi XQuery) in formato stringa che viene eseguita dalla la funzione `eval_querystring` definita nel modulo `Toputils` di Galax. Le query eseguite da Galax ricercano nei file dei metadati relativi agli URI specificati i valori delle proprietà RDF corrispondenti ai tre costrutti. Un esempio di espressione XQuery eseguita dal comando `use` è riportato di seguito:

```
namespace h=\"http://www.cs.unibo.it/helm/schemas/mattone.rdf#\"
namespace rdf=\"http://www.w3.org/1999/02/22-rdf-syntax-ns#\"
 for $i in
  document(/projects/helm/metadata/backward/Algebra/fac.con.xml)
  /rdf:RDF/h:Object/h:bacKPointer/h:Occurrence
 return ($i/h:occurrence, $i/h:position)
```

Tale espressione seleziona i riferimenti "backward" dell'oggetto `cic:/Algebra/fac.con` presenti nel file di metadati `/projects/helm/metadata/backward/Algebra/fac.con.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<rdf:RDF xml:lang="en"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:h="http://www.cs.unibo.it/helm/schemas/mattone.rdf#">
  <h:Object rdf:about="cic:/Algebra/fac.con">
    <h:backPointer>
      <h:Occurrence>
        <h:position>InConclusion</h:position>
        <h:occurrence>
          cic:/Algebra/nat_fac_gtzero.con
        </h:occurrence>
      </h:Occurrence>
    </h:backPointer>
    <h:backPointer>
      <h:Occurrence>
        <h:position>InConclusion</h:position>
        <h:occurrence>
```

```
        cic:/Algebra/nring_fac_ap_zero.con
        </h:occurrence>
      </h:Occurrence>
    </h:backPointer>
  </h:Object>
</rdf:RDF>
```

In particolare produce le coppie:

```
(InConclusion, cic:/Algebra/nat\_fac\_gtzero.con)}
(InConclusion, cic:/Algebra/nring_fac_ap_zero.con)}
```

I risultati restituiti da Galax vengono poi adattati al formato usato dagli altri comandi dell'interprete usando le funzioni delle librerie `List` e `String` di Ocaml.

4.3.1 Prestazioni

In questa sezione vengono riportati i risultati dei test delle prestazioni dell'interprete. Le prove sono state effettuate su esempi di query simili a quelle generate dai moduli del proofchecker per i termini Cic che interrogano la libreria utilizzando l'interprete. In particolare si sono utilizzati esempi di query costruite automaticamente dal modulo `MQueryGenerator`: a partire da insiemi di vincoli ottenuti ispezionando i termini Cic della dimostrazione produce una query (funzione `searchPattern()`) i cui risultati siano tali da soddisfare i vincoli. Nelle prime prove effettuate ci si è resi conto che l'esecuzione delle query viene notevolmente rallentata dalla fase di type-checking statico delle query che, come già anticipato, viene effettuata da Galax nel momento in cui la query viene "mappata" nel Datamodel prima dell'esecuzione. Ad esempio la query:

```
let %universe be reference
'cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)',
'cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1/1)'
in
select uri
in
(select ref1 in
```

```
use reference 'cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)'
position $str1
where $str1 is inconclusion)
intersect
(select ref2 in
use reference 'cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1/1)'
position $str2
where $str2 is inconclusion)
where
(select uri2 in
usedby uri
position $pos
where ($pos is inconclusion or $pos is mainconclusion )
subset %universe)
```

implica l'esecuzione di query XQuery simili a quella vista in precedenza ad ogni invocazione dei costrutti `use` e `usedby`. I tempi di esecuzione, misurati con la funzione `Unix.time()` di Ocaml, erano superiori al minuto.

Poiche' le query XQuery vengono generate automaticamente ed hanno sempre lo stesso formato, indipendente dai dati sui quali vengono eseguite, si è pensato di disabilitare il type-checking statico e di eseguirlo solo manualmente per verificare che le query contengano espressioni con il tipo opportuno.

I risultati ottenuti nelle successive sperimentazioni mostrano che i tempi di esecuzione delle query sono confrontabili con i tempi ottenuti con l'interprete che usa il database anche se in tutti i casi testati l'appoccio basato sul DBMS risulta vincente.

Num. vincoli	**Num. risultati**	**Interprete Postgres**	**Interprete Galax**
4	2	4 sec.	4 sec.
5	3	4 sec.	5 sec.
5	4	1 sec.	2 sec.
7	1	2 sec.	3 sec.
7	1	2 sec.	3 sec.
8	1	1 sec.	2 sec.
8	2	1 sec.	3 sec.
8	4	1 sec.	2 sec.
8	1	1 sec.	1 sec.
9	1	1 sec.	3 sec.

Capitolo 5

Linguaggio Mathql

L'evoluzione e la generalizzazione del linguaggio descritto nel capitolo precedente ha portato alla definizione di un vero e proprio linguaggio per l'interrogazione di metadati basati sullo standard RDF.

Alcune delle motivazioni che ci hanno indotto a modificare la semantica del linguaggio Mathql sono accennate in [Lor02]: nella parte conclusiva di tale documento di tesi, viene messa in evidenza l'esigenza di predisporre il sistema di gestione di metadati in modo da supportare documenti RDF di origine qualsiasi, e non solo documenti basati sullo schema delle dipendenze "forward/backward". Da tale constatazione deriva una serie di modifiche riguardanti l'introduzione di tipi di dato più "robusti", in modo da gestire anche gli schemi di metadati RDF più articolati, e di costrutti appropriati, cioè tali da gestire im modo più efficiente le operazioni sulle proprietà RDF degli oggetti del sistema. In particolare il costrutto `intersect`, come verrà chiarito meglio in 5.2, ora getisce in modo appropriato gli insiemi di attributi associati agli URI degli oggetti.

Come dettagliato meglio in seguito, gli oggetti corrispondenti agli output delle query sono degli insiemi di risorse in cui ciascuna risorsa è costituita da una URI reference e da un insieme di attributi (di tipo stringa multipla) organizzati in gruppi in base al contesto di appartenenza (gruppo dei metadati Dublin Core, metadati di Eulero,..). Gli attributi oltre a contenere informazioni sulle risorse alle quali sono associati (ad es. metadati Dublin Core) descrivono relazioni tra risorse rispecchiando in modo naturale il concetto astratto di proprietà nel modello di dati di RDF.

5.1 Tipi di dato

5.1.1 Tipi primitivi

Il tipo dei valori booleani è indicato con `Boole` e i suoi elementi sono `T` e `F`. Le stringhe sono rappresentate dal tipo `String` e i suoi elementi sono sequenze finite di caratteri Unicode (vedi??). Le produzioni grammaticali, rappresentate in parentesi angolose, sono usate per denotare il sottotipo di `String` contenente le sequenze di caratteri prodotte.

5.1.2 Costruttori di tipi

Il costruttore `Setof Y` indica il tipo degli insiemi finiti (sequenze finite e non ordinate) di elementi in `Y`, senza ripetizioni. `Listof Y` denota il tipo lista (cio sequenze finite e ordinate) di elementi in `Y`. In seguito si user la notazione $[y_1, \ldots, y_m]$ per la lista i cui elementi sono $y_1, \ldots, y_m$. Partendo da questi costruttori possiamo definire i seguenti operatori su insiemi:

- $\emptyset$: (`Setof Y`)
- $\exists$: ((`Setof Y`) $\rightarrow$ `Boole`) $\rightarrow$ `Boole`
- $\in$: `Y` $\rightarrow$ (`Setof Y`) $\rightarrow$ `Boole` (infisso)
- $\subseteq$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ `Boole` (infisso)
- $\between$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ `Boole` (infisso)
- $\cap$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ (`Setof Y`) (infisso)
- $\cup$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ (`Setof Y`) (infisso)
- $\sqcup$: (`Setof Y`) $\rightarrow$ (`Setof Y`) $\rightarrow$ (`Setof Y`) (infisso, unione disgiunta)

  $U \between W$ indica $(\exists\, u \in U)\ u \in W$ come spiegato in dettaglio in[GS]. $Y * Z$ indica il prodotto dei tipi Y e Z e i suoi elementi sono le coppie (y, z) tali che $y : Y$ e $z : Z$. $Y \rightarrow Z$ indica il tipo delle funzioni da Y a Z e fy indica l'applicazione di $f : Y \rightarrow Z$ a $y : Y$. Le relazioni sui tipi, come l'uguaglianza, sono viste come funzioni a valori in `Boole`.

- `Fst`:(`Y` $\times$ `Z`) $\rightarrow$ `Y` tale che `Fst`$(y, z) = y$.

- $\texttt{FstSet}:(\texttt{Setof}(\texttt{Y}\times\texttt{Z})) \to (\texttt{Setof Y}$ tale che $\texttt{FstSet}W = \{\texttt{Fst}w \mid w \in W\}$
- $W(y) : (\texttt{Setof}(\texttt{Y} \times \texttt{Z})) \to \texttt{Y} \to \texttt{Z}$, nel caso in cui W contenga una sola coppia il cui primo elemento è y. Negli altri casi $W(y)$ non è definito.
- $W[y \to z] : (\texttt{Setof}(\texttt{Y}\times\texttt{Z})) \to \texttt{Y} \leftarrow \texttt{Z} \to (\texttt{Setof}(\texttt{Y}\times\texttt{Z}))$ è l'insieme ottenuto da W eliminando ogni coppia il cui primo elemento è y e aggiungendo la coppia (y, z).
- $U + W$ è l'unione di due insiemi di coppie come segue:

$$U + \emptyset \qquad \text{diventa} \qquad U$$

$$U + (W \sqcup \{(y, z)\}) \qquad \text{diventa} \qquad U[y \leftarrow z] + W$$

  Gli ultimi tre operatori sono usati per manipolare gli association set cioè insiemi di coppie tali che le prime componenti di due coppie distinte sono sempre diverse. Tali insiemi saranno usati per formalizzare i contesti di valutazione. I tipi degli oggetti del linguaggio sono:

- Una stringa multipla V è un oggetto con tipo T_0 `= Setof String`.
- Un path, usato per specificare il nome di una risorsa eventualmente strutturata, è una lista non nulla in cui il primo elemento è sempre presente: ha tipo `String` $\times$ `Setof String`.
- Gli insiemi B nelle triple delle relazioni tra risorse sono degli association set con tipo: `Setof` $(T_5 \times T_0)$.
- Un gruppo di attributi G è un association set che mette in relazione i nomi di attributi ai loro valori, il tipo è T_1 `= Setof(String`$\times T_0$`)`.
- Una reference r è un oggetto di tipo `String`.
- Un insieme A di gruppi di attributi di una risorsa ha tipo T_2 `= Setof` T_1.
- Una risorsa è una reference a cui sono associati gruppi di attributi, cioè un oggetto di tipo T_3 `= (String` $\times T_2$`)`.
- Il tipo di un insieme di risorse è T_4 `= Setof` T_3.

Nella sezione seguente verranno descritte in dettaglio la semantica e la sintassi del linguaggio, in particolare negli esempi un insieme di risorse S sarà rappresentato come segue:

```
S: "uri1" {"A" = "a"}; "uri2" {"B" = "b1","b2"; "C" = "c"};
   "uri3" {"D" = "d"}; "uri4" {"E" = "e"}, {"F" = "f"}
```

in cui ad esempio a uri2 è associato un gruppo contenente gli attributi B (con valori b1,b2) e C (con valore c). A uri4, invece, sono associati due gruppi: uno contenente l'attributo E e l'altro l'attributo F.

Le funzioni primitive, di cui si parla diffusamente in seguito, usate per manipolare le informazioni che il query engine implementato ottiene dal database sottostante sono: Attribute, Fun, Relation, Rename, Pattern, Unquote.

5.2 Sintassi testuale e semantica di MathQL

Le espressioni del linguaggio si dividono in tre categorie:

- Espressioni che denotano insiemi di risorse, esse appartengono alla produzione grammaticale <mq_set> e la loro semantica è data dalla relazione di valutazione infissa $\Downarrow_s$.
- Espressioni che denotano condizioni booleane, si ottengono con la produzione grammaticale <mq_boole> e la loro semantica è data dalla relazione di valutazione infissa $\Downarrow_b$.
- Le espressioni che denotano un valore di stringa multipla appartengono alla produzione grammaticale <mq_val> e la rispettiva relazione di valutazione è $\Downarrow_v$

Le espressioni possono contenere stringhe tra virgolette (quoted string), la cui sintassi è:

```
<hex>         ::= '0 -9' |'A - F' | 'a - f']
<escaped>     ::= "u" <hex> <hex> <hex> <hex> | '"' | "\" | "^"
<string>      ::= '"' [ "\" <escaped> "^" | '^ "\^' ]* '"'
<string_list> ::= <string> [ "," <string> ]*
<path>        ::= <string> [ "/" <string> ]*
```

La stringa racchiusa tra le virgolette viene estratta seguendo la tabella 5.1

Escape sequence	**Unicode character**	**Text**
\u....^	U+....	
\"^	U+0022	"
\\^	U+005C	\
\ ^^	U+005E	^

Tabella 5.1: Tabella Unicode

L'operazione di estrazione viene eseguita da `Unquote` che ha tipo: `String` → `String`. Le query possono anche contenere variabili da associare alle risorse (rvar), variabili da associare agli insiemi di risorse (svar) e variabili per le stringhe multiple (vvar).

```
<alpha>     ::= [ 'A - Z' | 'a - z']+
<number>    ::= [ '0 - 9' ]+
<id>        ::= <alpha> [ <alpha> | <number> ]*
<mq_svar> ::= "@" <id>
<mq_rvar> ::= "%" <id>
<mq_vvar> ::= "$" <id>
```

Il contesto di valutazione delle espressioni $\Gamma = (\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v)$ è una quadrupla di association set: Γ_s ha tipo K_s:`Setof (<mq_svar> ×` T_4`)` e lega le svar agli insiemi di risorse; Γ_r ha tipo K_r: `Setof (<mq_rvar> ×` T_3`)`, associa le rvar alle risorse; Γ_a di tipo: K_a: `Setof (<mq_rvar> ×` T_1`)`, lega le rvar ai gruppi di attributi; Γ_v con tipo K_v: `Setof (<mq_vvar> ×` T_0`)`, mette in relazione le vvar con i valori di stringhe multiple.
Il tipo K di Γ è: $K_s \times K_r \times K_a \times K_v$

e i tipi delle relazioni di valutazione:

$\Downarrow_s$:(K × <mq_set>) → T_4 → Boole,
$\Downarrow_b$:(K × <mq_boole>) → Boole → Boole,
$\Downarrow_v$:(K × <mq_val>) → T_0 → Boole.

Espressioni che denotano insiemi di risorse Queste espressioni rappresentano query o sottoparti di esse.

```
<refine>    ::= [ "sub" | "super" ]?
<qualifier> ::= [ "inverse" ]? <refine> <path>
<assign>    ::= <mq_vvar> "<-" <path>
<attr_list> ::= [ "attr" <assign> ["," <assign> ]* ]?
<mq_set>    ::= "ref" <mq_val>
              | "pattern" <mq_val>
              | <mq_svar> | <mq_rvar>
              | "(" <mq_set> ")"
              | "relation" <qualifier> <mq_set> <attr_list>
              | "select" <mq_rvar> "in" <mq_set> "where" <mq_boole>
              | <mq_set> "union" <mq_set>
              | <mq_set> "intersect" <mq_set>
              | <mq_set> "diff" <mq_set>
              | "let" <mq_svar> "be" <mq_set> "in" <mq_set>
              | "let" <mq_vvar> "be" <mq_val> "in" <mq_set>
```

intersect, union and diff sono associative a sinistra e hanno ordine di precedenza decrescente.

Il costrutto `let` lega rispettivamente una svar ad un insieme di risorse nel contesto Γ_s ed una vvar ad un insieme di stringhe nel contesto Γ_v, formalmente:

$$\frac{i: \texttt{<mq_svar>} \quad ((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), x_1) \Downarrow_s S_1 \qquad ((\Gamma_s[i \leftarrow S_1], \Gamma_r, \Gamma_a, \Gamma_v), x_2) \Downarrow_s S_2}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), \text{ let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_s S_2}$$

$$\frac{i: \texttt{<mq_vvar>} \quad ((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), x_1) \Downarrow_v V \qquad ((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v[i \leftarrow V]), x_2) \Downarrow_s S}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), \text{ let } i \text{ be } x_1 \text{ in } x_2) \Downarrow_s S}$$

Ad esempio con l'espressione:

```
let %s be E1 in E2
```

la variabile `%s` viene legata all'insieme di risorse restituite dalla valutazione dell'espressione `E1` nel contesto delle svar, e con tale contesto viene valutata l'espressione `E2`.

I costrutti per leggere le variabili e per la parentesizzazione sono:

$$\frac{i:\ \texttt{<mq_svar>}}{((\Gamma_s,\Gamma_r,\Gamma_a,\Gamma_v),i)\Downarrow_s \Gamma_s(i)} \quad \frac{i:\ \texttt{<mq_rvar>}}{((\Gamma_s,\Gamma_r,\Gamma_a,\Gamma_v),i)\Downarrow_s \Gamma_r(i)} \quad \frac{(\Gamma,x)\Downarrow_s S}{(\Gamma,(x))\Downarrow_s S}$$

$\Gamma_s(i)$ e $\{\Gamma_r(i)\}$ valutano $\emptyset$ se i non è definita.

Il costrutto `union` ”somma” due insiemi di risorse costruendo l'unione dei due insiemi: le risorse che compaiono in entrambi gli insiemi (risorse con lo stesso URI) vengono collassate in un unica risorsa con tutti i gruppi di attributi delle due risorse (se vi sono gruppi identici vengono presi una sola volta). Formalmente:

$$\frac{(\Gamma,x_1)\Downarrow_s S_1 \quad (\Gamma,x_2)\Downarrow_s S_2}{(\Gamma,x_1 \text{ union } x_2)\Downarrow_s S_1\oplus S_2}$$

dove $\oplus$ mette insieme i gruppi di attributi appartenenti alla stessa risorsa eliminando eventuali valori doppi:

1	$(S_1 \sqcup \{(r,A_1)\}) \oplus (S_2 \sqcup \{(r,A_2)\})$	diventa	$S_1 \oplus S_2 \oplus \{(r, A_1 \cup A_2)\}$
2	$S_1 \oplus S_2$	diventa	$S_1 \cup S_2$

$\oplus$ è definito associativo e la regola 1 ha precedenza sulla regola 2 Se, ad esempio, la valutazione delle espressioni `E1` e `E2` restituisce gli insiemi `S1`e `S2`:

```
S1: "uri1" {"A" = "a"}; "uri2" {"B" = "b1"}
S2: "uri2" {"B" = "b1"}, {"C" = "c"}; "uri4" {"E" = "e"}
```

allora l'espressione:

```
E1 union E2
```

restituirà:

```
S3: "uri1" {"A" = "a"}; "uri2" {"B" = "b1"}, {"C" = "c"};
    "uri4" {"E" = "e"}
```

Il costrutto `intersect` ”moltiplica” due insiemi di risorse nel seguente modo:

$$\frac{(\Gamma,x_1)\Downarrow_s S_1 \quad (\Gamma,x_2)\Downarrow_s S_2}{(\Gamma,x_1 \text{ intersect } x_2)\Downarrow_s S_1\otimes S_2}$$

dove $\otimes$ dà il prodotto di gruppi di attributi appartenenti alla stessa risorsa:

1	$(S_1 \sqcup \{(r, A_1)\}) \otimes (S_2 \sqcup \{(r, A_2)\})$	diventa	$(S_1 \otimes S_2) \cup \{A_1 \times A_2)\}$
2	$S_1 \otimes S_2$	diventa	$\emptyset$

la regola 1 ha precedenza sulla regola 2 e $A_1 \times A_2 = \{G_1 \oplus G_2 | G_1 \in A_1, G_2 \in A_2\}$ L'insieme di risorse restituito contiene tutte le risorse contenute in entrambi gli insiemi: i gruppi di attributi delle risorse dell'insieme risultato sono dati dal prodotto cartesiano dei gruppi di attributi delle risorse (con lo stesso URI) nei due insiemi. Ad esempio se le espressioni `E1` e `E2` producono gli insiemi:

```
S4: "uri1" {"A" = "a"},{"B" = "b1"}
S5: "uri1" {"A" = "a"}, {"B" = "b2"}
```

allora

```
E1 intersection E2
```

darà:

```
S6: "uri1" {"A" = "a"},{"B" = "b1", "b2"},
           {"A" = "a"; "B" = "b2"}, {"B" = "b1"; "A" = "a"}
```

Il costrutto `diff` fa la "differenza" tra due insiemi di risorse come segue:

$$\frac{(\Gamma, x_1) \Downarrow_s S_1 \quad (\Gamma, x_2) \Downarrow_s S_2}{(\Gamma, x_1 \text{ diff } x_2) \Downarrow_s S_1 \ominus S_2}$$

dove $A \ominus B$ prende solo le risorse di A, con i corrispondenti insiemi di attibuti, che non appartengono a B senza considerare gli attributi associati:

1	$(S_1 \sqcup \{(r, A_1)\}) \ominus (S_2 \sqcup \{(r, D_2)\})$	diventa	$(S_1 \ominus S_2)$
2	$S_1 \ominus S_2$	diventa	S_1

La regola 1 ha precedenza sulla regola 2.

Applicando il comando `diff` agli insiemi `S1` e `S2` visti in precedenza, si avrà:

```
S7: "uri1" {"A" = "a"}
```

Il costrutto `ref` (reference) trasforma un insieme di reference in un insieme di risorse, ognuna senza attributi (cioè trasforma il tipo T_0 nel tipo T_4):

$$\frac{(\Gamma, x) \Downarrow_v V}{(\Gamma,\ \text{ref}\ x) \Downarrow_s \{(s, \emptyset) \quad | \quad s \in V\}}$$

Il costrutto `pattern` denota l'insieme di risorse (senza attributi) le cui reference fanno match con almeno una delle espressioni regolari `POSIX` 1003.2 – 1992[1] date:

$$\frac{(\Gamma, x) \Downarrow_v V}{(\Gamma,\ \text{pattern}\ x) \Downarrow_s \{(r, \emptyset) \quad | \quad (\exists s \in V) \quad r \in \ \text{Pattern}\ s\}}$$

dove `Pattern` s rappresenta l'insieme di reference che fanno match con l'espressione regolare s, tra quelle disponibili nella base di dati.

Il costrutto `relation` costruisce l'insieme di risorse nella relazione specificata (r_1, r_2, B), nelle sue sub-relazioni (`sub`) o nelle sue super-relazioni (`super`) con una o più risorse in un dato insieme. Ognuna di queste risorse ha l'insieme di attributi composto dagli attributi specificati negli assegnamenti che seguono la clausola "`attr`".

$$\frac{f : \texttt{<refine>} \quad p : \texttt{<path>} \quad (\Gamma, x) \Downarrow :_s S \quad q_1 : \texttt{<assign>} \ldots q_m : \texttt{<assign>}}{(\Gamma, \text{relation}\ f\ p\ x\ \text{attr}\ q_1, \ldots, q_m) \Downarrow_s \bigoplus\{\{(r_2, D_2)\} \mid (\exists (r_1, D_1)\ \in\ S)\ (r_1, r_2, B)\ \in\ R\}}$$

dove D_2 è $\{$`Assign` B $q_1 \ldots q_m\}\}$ e R è `Relation` f (`Name`p). In particolare `Assign` costruisce un insieme (eventualmente composto) di attributi i cui valori sono presi da B usando le corrispondenze date da $q_1 \ldots q_m$. Formalmente se i:`<vvar>` e p:`<path>`, definiamo `Assign` $B\ Q = \oplus\{(i, B(\texttt{Name}p)) | (i \leftarrow p) \in Q\}$. `Relation` fs restituisce l'insieme delle triple (R) della relazion s, quando s è un path di stringhe $s_1/\ldots/s_n$ `Relation` restituisce le triple della relazione ottenuta dalla composizione di $s_1 \ldots s_n$ (ad esempio nel caso di una proprietà strutturata s_1 descritta, a sua volta da una proprietà s_2 ...).
Se f è `sub` o `super`, `Relation` restituisce anche le triple rispettivamente delle

[1] `Incluse in POSIX 1003.1-2001: <http://www.unix-systems.org/version3/iee_std.html>.`

sub-relazioni o super-relazioni della relazione s.
Se è presente la clausola `inverse`, viene usata la relazione inversa: l'argomento x invece di essere il soggetto della proprietà, come avviene nella relazione diretta, rappresenta l'oggetto della proprietà. Formalmente sostituiamo R con $R' = \{(r1, r2, B) \mid (r2, r1, B) \in R\}$ nella regola precedente. Si noti che le `vvar` introdotte dal costrutto `relation` differiscono dalle `vvar` introdotte dal costrutto `let`, persino se hanno lo stesso nome.

Per chiarire meglio il comportamento del costrutto `relation` facciamo un esempio usando i metadati della libreria elettronica del progetto Helm descritti in 2.3. L'espressione:

```
relation "refObj" "cic:/Coq/Init/Peano/le.ind#1/1"
  attr $p<-"position"
```

restituisce l'insieme delle risorse nella relazione `refObj` con la risorsa identificata dall'URI `cic:/Coq/Init/Peano/le.ind#1/1`. A tali risorse è associato l'attributo `$p` a cui è assegnato il valore della sottoproprietà `position` di `refObj`.

Il costrutto `select` estrae da un insieme di risorse quelle che soddisfano la condizione booleana data. La condizione è testata per ogni risorsa dell'insieme e le risorse sono assegnate a una rvar prima del test.

$$\frac{i : \texttt{<rvar>} \quad (\Gamma, x_1) \Downarrow_s S}{(\Gamma,\ \text{select } i \text{ in } x_1 \text{ where } x_2) \Downarrow_s \ \texttt{Select}\ S\, i\, \Gamma\, x_2}$$

dove la funzione `Select` è definita come segue:

$$\frac{i : \texttt{<rvar>}}{\texttt{Select}\ \emptyset\, i\, \Gamma\, x \text{ diventa } \emptyset}$$

$$\frac{i : \texttt{<rvar>} \quad ((\Gamma_s, \Gamma_r[i \leftarrow R], \Gamma_a, \Gamma_v), x) \Downarrow_b F}{\texttt{Select}(S \sqcup \{R\})\, i\, \Gamma\, x \text{ diventa } \texttt{Select} S\, i\, \Gamma\, x}$$

$$\frac{i : \texttt{<rvar>} \quad ((\Gamma_s, \Gamma_r[i \leftarrow R], \Gamma_a, \Gamma_v), x) \Downarrow_b T}{\texttt{Select}(S \sqcup \{R\})\, i\, \Gamma\, x \text{ diventa } (\texttt{Select} S\, i\, \Gamma\, x) \cup R}$$

dove R è una coppia della forma (r, A) e $\Gamma = (\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v)$ Usando il costrutto `select`, dall'insieme di risorse ottenuto nell'esempio del costrutto `relation`, si

può estrarre un sottoinsieme i cui elementi soddisfano determinate condizioni. Ad esempio, la query:

```
select @uri in
  relation "refObj" "cic:/Coq/Init/Peano/le.ind#1/1"
    attr $p<-"position"
where
ex("MainConclusion" sub @uri.$p)
```

restituisce tutte le risorse dell'insieme ottenuto nell'esempio precedente che soddisfano la condizione booleana specificata nella clausola `where` (l'uso dei costrutti `ex` e "." saranno chiariti in seguito). Dal punto di vista operativo, le risorse nell'insieme generato dalla valutazione di `relation` vengono legate (una per volta) alla rvar `@uri` nel contesto delle rvar e con tale contesto si valuta l'espressione booleana.

Espressioni che denotano una condizione booleana Queste espressioni vengono usate nella clausola `where` del costrutto `select`.

```
<mq_boole> ::= "false" | "true"
             | "(" <mq_boole> ")"
             | "not" <mq_boole> | "ex" <mq_boole>
             | <mq_boole> "and" <mq_boole>
             | <mq_boole> "or" <mq_boole>
             | <mq_val> "sub" <mq_val>
             | <mq_val> "meet" <mq_val>
             | <mq_val> "eq" <mq_val>
```

and e *or* sono associativi a sinistra.
L'ordine di precedenza (decrescente) è: not, and, or, ex.

Alcuni costrutti per costanti, operazioni standard e parentesizzazione:

$$\frac{}{(\Gamma, \texttt{false}) \Downarrow_b \texttt{F}} \quad \frac{}{((\Gamma, \texttt{true}) \Downarrow_b \texttt{T}} \quad \frac{(\Gamma, x) \Downarrow_b b}{(\Gamma, (x)) \Downarrow_b b} \quad \frac{(\Gamma, x) \Downarrow_b \texttt{T}}{(g, \texttt{not}\ x) \Downarrow_b \texttt{F}} \quad \frac{(\Gamma, x) \Downarrow_b \texttt{F}}{(g, \texttt{not}\ x) \Downarrow_b \texttt{T}}$$

$$\frac{(\Gamma, x_1) \Downarrow_b \texttt{F}}{(\Gamma, x_1\ \texttt{and}\ x_2) \Downarrow_b \texttt{F}} \quad \frac{(\Gamma, x_1) \Downarrow_b \texttt{T} \quad (\Gamma, x_2) \Downarrow_b b}{(\Gamma, x_1\ \texttt{and}\ x_2) \Downarrow_b b}$$

$$\frac{(\Gamma, x_1) \Downarrow_b \texttt{T}}{(\Gamma, x_1 \ \texttt{or} \ x_2) \Downarrow_b \texttt{T}} \quad \frac{(\Gamma, x_1) \Downarrow_b \texttt{F} \quad (\Gamma, x_2) \Downarrow_b b}{(\Gamma, x_1 \texttt{or} x_2) \Downarrow_b b}$$

Si nota che le operazioni binarie sono valutate con una strategia early-out.

Il costrutto `ex` (esiste) dà accesso agli insiemi di attributi associati agli insiemi di risorse contenuti nella parte Γ_r del contesto Γ e ciò è ottenuto caricando la parte Γ_a che è usata dal costrutto `<mq_rvar>.<mq_vvar>` descritto nella prossima sezione. `ex` restituisce true se la condizione che lo segue è soddisfatta da almeno un pool di insiemi di attributi, uno per ogni risorsa nella parte Γ_r del contesto. Formalmente:

$$\overline{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), ex\ x) \Downarrow_b ((\exists \Delta_a \ \in \texttt{All}\Gamma_r)\ ((\Gamma_s, \Gamma_r, \Gamma_a + \Delta_a, \Gamma_v),\ x) \Downarrow_b T)}$$

dove $\texttt{All}\ \ \Gamma_r = \{\Delta_a | \Delta_a(i) = A \text{ iff } \Gamma_r(i) = (r, D \cup \{A\})\}$ e Δ_a ha lo stesso tipo di Γ_a.

I costrutti `sub,` `meet` e `eq` confrontano due insiemi di stringhe: `sub` verifica che le stringhe contenute nel primo insieme siano anche nel secondo; `meet` controlla se tra le stringhe appartenti al primo insieme ve ne sia almeno una anche nel secondo insieme; `eq` verifica l'uguaglianza tra i due insiemi dati. I confronti sono estensionali e l'uguaglianza tra stringhe è case-sensitive.

$$\frac{(\Gamma, x_1) \Downarrow_v V_1 \quad (\Gamma, x_2) \Downarrow_v V_2}{(\Gamma, x_1\ sub\ x_2) \Downarrow_b (V_1 \subseteq V_2)} \quad \frac{(\Gamma, x_1) \Downarrow_v V_1 \quad (\Gamma, x_2) \Downarrow_v V_2}{(\Gamma, x_1\ meet\ x_2) \Downarrow_b (V_1 \between V_2)}$$

$$\frac{(\Gamma, x_1) \Downarrow_v V_1 \quad (\Gamma, x_2) \Downarrow_v V_2}{(\Gamma, x_1\ eq\ x_2) \Downarrow_b (V_1 = V_2)}$$

L'operatore `eq` è introdotto poiché la valutazione di $x_1\ eq\ x_2$ può essere più efficiente di quella di $x_1\ sub\ x_2$ e $x_2\ sub\ x_1$.
Ad esempio se le espressioni `e1` e `e2` producono gli insiemi:

```
V1:{"a", "b", "c", "d"}
V2:{"e", "c", "g", "h"}
```

allora

```
e1 sub e2
```

restituisce `false`, mentre

```
e1 meet e2
```

ritorna `true`, e ovvaimente

```
e1 eq e2
```

da `false`

Espressioni che denotano stringhe multiple Queste espressioni compaiono come operandi dei costrutti `sub, meet` e `eq`.

```
<mq_val> ::= <string> | "{" [ <string> [ "," <string> ]* ]? "}"
           | "refof" <mq_set>
           | <mq_vvar> | <mq_rvar> "." <mq_vvar>
           | "(" <mq_val> ")"
           | "property" <qualifier> <mq_val>
```

Un insieme di stringhe può essere ottenuto esplicitamente con i seguenti costrutti:

$$\frac{q_1 : \texttt{<string>} \ \ldots \ \ q_m : \texttt{<string>}}{(\Gamma, \{q_1, ..., q_m\}) \Downarrow_v \{ \ \texttt{Unquote} \ q_1, ..., \ \texttt{Unquote} \ q_m\}} \quad \frac{q : \texttt{<quoted_string>}}{(\Gamma, q) \Downarrow_v \{\texttt{Unquote} \ q\}}$$

Il costrutto `refof` (reference of) consente di ottenere un insieme di stringhe estraendole da un insieme di risorse (trasforma il tipo T_4 nel tipo T_0):

$$\frac{(\Gamma, x) \Downarrow_s S}{(\Gamma, \texttt{refof} \ \ x) \Downarrow_v (\texttt{FstSet} \ \ S)}$$

Il costrutto `property` ha una semantica molto simile a quella di `relation` e consente di estrarre i valori di una data proprietà, o sotto-proprietà o super-proprietà, associati ad un insieme di stringhe corrispondenti agli oggetti della proprietà data. Tale costrutto è usato effettuare controlli sui valori delle proprietà al fine di "filtrare" i risultati di altre espressioni. Formalmente:

$$\frac{f : \texttt{<refine>} \quad p : \texttt{<path>} \quad (\Gamma, x) \Downarrow_v V}{(\Gamma, \text{property } f\, p\, x) \Downarrow_v \{v | (\exists r \in V)\ (r, v) \in Q\}}$$

dove $Q =$ `Property` f (`Name`p) e `Property` f s produce l'insieme delle coppie della proprietà s che rappresentano i valori delle istanze delle proprietà attribuite agli elementi referenziati dal primo elemento di una coppia. Se $s = (s_0, [s_1, ..., s_m])$, `Property` restituisce l'insieme di coppie della proprietà ottenute per composizione delle proprietà $s_0, ..., s_m$. Come se, ad esempio, una proprietà s_0 è descritta da un'altra proprietà s_1, ... Se f è `sub` o `super`, `Property` restituisce anche le coppie della sotto o super-proprietà di s. Se è presente la clausola `inverse` allora l'argomento x rappresenta l'oggetto della proprietà anzichè il soggetto (come avviene in assenza della clausola `inverse`). Formalmente, nella regola precedente Q viene sostituito con $Q' = \{(v, r) \mid (r, v) \in Q\}$.

Ritornando ai metadati di Helm, con l'espressione:

```
property "refObj"/"position" "cic:/Coq/Init/Peano/le.ind#1/1"
```

si ottiene la lista dei valori assunti dalla sottoproprietà `position` della relazione `refObj` per la risorsa identificata da `cic:/Coq/Init/Peano/le.ind#1/1`

Abbiamo inoltre due costrutti per leggere le vvar: uno legge la parte Γ_a del contesto che è aggiornata dalla clausola `ex`, mentre l'altro legge la parte Γ_v che è aggiornata dal costrutto `let` per le vvar:

$$\frac{i_1 : \texttt{<mq_rvar>} \quad i_2 : \texttt{<mq_vvar>}}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), i1.i2) \Downarrow_v \Gamma_a(i_1)(1_2)} \qquad \frac{i : \texttt{<mq_vvar>}}{((\Gamma_s, \Gamma_r, \Gamma_a, \Gamma_v), i) \Downarrow_v \Gamma_v(i)}$$

$\Gamma_a(i_1)(i_2)$ e $\Gamma_v(i)$ sono uguali a $\emptyset$ se i_1,i_2 o i non sono definiti.
Il comando "." si può usare in `select` per esprimere condizioni sugli attributi delle risorse legate alle rvar da quest'ultimo costrutto, ad esempio in:

```
select @uri in
  relation "refObj" "cic:/Coq/Init/Peano/le.ind#1/1"
    attr $p<-"position"
where
  ex("MainConclusion" sub @uri.$p)
```

le risorse restituite dal comando `relation` vengono legate a `@uri` e, nella clausola `where`, si fa riferimento agli attributi associati a tali risorse.
Purtroppo quando si vuole leggere una vvar si è costretti a specificare anche la rvar associata, in compenso, però, tale modo di operare ci consente di evitare le ambiguità che potrebbero verificarsi nei casi di vvar con lo stesso nome associate a rvar distinte, come nell'espressione che segue:

```
let %s be
  relation "refObj" "cic:/Coq/Init/Peano/le.ind#1/1"
    attr $p<-"position"
in
  select @uri in %s
  where
    "MainConclusion"
    sub
    refof
      select @r2 in %s
      where
        ex (@r1.$a sub @r2.$a)
```

5.3 Sintassi testuale e semantica dei risultati delle query

La rappresentazione testuale dei risultati delle query appartiene alla produzione grammaticale `<mq_set>` la cui semantica è rappresentata dalle relazioni: $\Rightarrow_a, \Rightarrow_g, \Rightarrow_r, \Rightarrow_s$.

```
<mqr_attr>  ::= <string> [ "=" <string> [ "," <string>]* ]?
<mqr_group> ::= "{" <mqr_attr> [ ";" <mqr_attr> ]* "}"
<mqr_res>   ::= <string>
                [ "attr" <mqr_group> [ "," <mqr_group> ]* ]?
<mqr_set>   ::= [ <mqr_res> [ ";" <mqr_res> ]* ]?
```

$\Rightarrow_a$ valuta un attributo che è di tipo stringa multipla. Il tipo di questa relazione è `<mqr_attr>` $\rightarrow$ `(String` $\times T_0$`)` $\rightarrow$ `Boole` e la valutazione avviene

nel modo seguente:

$$\frac{q : \texttt{<string>} \quad q_1 : \texttt{<string>} \; \ldots \quad q_m : \texttt{<string>}}{q = q_1, ..., q_m \Rightarrow_a (\texttt{Unquote}\ \ q, \{\texttt{Unquote}\ \ q_1, ..., \texttt{Unquote}\ \ q_m\})}$$

Si noti che una stringa multipla può essere vuota, ciò rispecchia il fatto che, in un modello di dati RDF, l'uso di una proprietà è opzionale persino se essa è sempre dichiarata in uno schema RDF.

$\Rightarrow_g$ valuta un insieme di attributi con i rispettivi valori. Il tipo è `<mqr_group>` $\rightarrow T_1 \rightarrow$ `Boole` e la regola di valutazione:

$$\frac{x_1 \Rightarrow_a a_1 ... x_m \Rightarrow_a a_m}{\{x_1, ..., x_m\} \Rightarrow_g \{a_1, ..., a_m\}}$$

Si noti che gli insiemi di attributi sono sempre inabitati quindi $m \geqslant 1$.

$\Rightarrow_r$ valuta le risorse con i rispettivi attributi. Il tipo è `<mqr_res>` $\rightarrow T_3 \rightarrow$ `Boole` e la valutzione avviene secondo la regola:

$$\frac{q : \texttt{<string>} \quad x_1 \Rightarrow_g A_1 \ldots x_m \Rightarrow_g A_m}{q\ \ \texttt{attr}\ \ x_1, ..., x_m \Rightarrow_r (\texttt{Unquote}\ \ q, \{A_1, ..., A_m\})}$$

$\Rightarrow_s$ valuta un insieme di risorse con i relativi insiemi di attributi. Il tipo è `<mqr_set>` $\rightarrow T_4 \rightarrow$ `Boole`

$$\frac{x_1 \ \Rightarrow_r \ r_1 \ldots \ x_m \Rightarrow_r r_m}{\{x_1, ..., x_m\} \Rightarrow_s \{r_1, ..., r_m\}}$$

5.4 Implementazione e sperimentazione nel progetto Helm

In questo lavoro di tesi è stato implementato il *back-end* per il linguaggio presentato nelle sezioni precedenti, in particolare si sono seguiti due approcci alternativi: uno si basa sull'utilizzo del DBMS `PostgreSQL`[PSQ] che, come verrà spiegato meglio in seguito, risulta essere quello più performante; l'approccio

alternativo consiste nell'utilizzo di un'implementazione di XQuery, *Galax*, per eseguire le query direttamente sui file dei metadati (in formato XML) senza il supporto di un DBMS.

Nelle sezioni seguenti vengono descritte le due implementazioni, è da notare che i due sistemi di interrogazione usano dei moduli in comune: le parti in cui si differenziano sono quelle relative rispettivamente alla ricerca dei metadati nella base di dati (interprete basato sull'uso del DBMS) e alla navigazione e attraversamento del file system dei metadati del progetto Helm (per il query engine basato su Galax).

5.4.1 Interprete basato su PostgreSQL DBMS

Per realizzare questo tipo di approccio si è scelto di usare il linguaggio Ocaml per favorire l'integrazione del software già sviluppato all'interno del progetto Helm.

L'interfacciamento al DBMS è stato realizzato usando il pacchetto *Postgres* [Posb] che consente l'esecuzione di query SQL e la gestione dei risultati generati mantenendo l'approccio funzionale e a oggetti del linguaggio Ocaml.

I moduli implementati si possono suddividere in due categorie: quelli relativi al reperimento delle informazioni richieste all'interno della base documentaria; e i moduli per la gestione e manipolazione delle strutture dati, usate per memorizzare i dati estratti dal database in un fromato interno, tali dati vengono elaborati in seguito per fornire i risultati delle query del linguaggio.

Le strutture dati utilizzate per implementare i tipi di dato descritti nella sezione precedente sono essenzialmente liste multiple di stringhe (tipo `string` di Ocaml). In particolare gli attributi delle risorse sono stati implementati con delle coppie formate da una stringa (nome dell'attributo) e da una lista (tipo `List` di Ocaml) di stringhe (valori che l'attributo può assumere). I gruppi di attributi sono delle liste di attributi e gli insiemi di gruppi di attributi sono delle liste di gruppi di attributi. Una risorsa è stata implementata con una coppia in cui il primo elemento è una stringa (URI della risorsa) e il secondo elemento è una lista di gruppi di attributi. Un insieme di risorse è una lista di risorse.

Il contesto utilizzato per memorizzare i risultati intermedi della valutazione delle query è un record (file `context.ml`) con quattro campi, ciascuno con tipo lista di coppie, che rappresentano rispettivamente: il contesto delle svar

(precedentemente riferito con Γ_s) che associa una svar ad una lista di risorse; il contesto delle rvar (Γ_r) che lega una rvar ad una risorsa; il contesto dei gruppi (Γ_a) in cui una rvar è legata ad un gruppo di attributi; il contesto delle vvar (Γ_v) usato per associare una vvar al valore di un attributo.

Organizzazione della base di dati e dettagli implementativi

Nonostante la nuova versione di Mathql sia molto più generale di quella precedente, dal punto di vista implementativo si è costretti a fornire delle soluzioni strettamente dipendenti dall'organizzazione dei metadati nel database che, come già detto, è stata ereditata dalla precedente implementazione.

Il formato di archiviazione, illustrato in maniera più dettagliata in [Lor02], prevede che le proprietà di un modello RDF siano memorizzate in tabelle separate per partizionare il più possibile l'insieme di dati della libreria elettronica per ottenere migliori prestazioni nelle operazioni di ricerca.

Le relazioni di dipendenza relative ad un modello specifico (proprietà RDF `refObj` e `backPointer`) sono rappresentate da una tabella con tre campi:

- `prop_id`: identificatore della proprietà;
- `context`: valore della posizione;
- `uri`: URI dell'oggetto referenziato.

prop_id	**context**	**uri**
F	InConclusion	cic:/Coc/Init/Logic/not.con
B	MainConclusion	cic:/Coc/Init/Logic/False.con
B	InBody	cic:/Algebra/Basics/NEG_anti_convert.con
F	InBody	cic:/Algebra/Basics/POS_anti_convert.con
...	...	...

Tabella 5.2: Esempio di relazione per un modello scpecifico: tabella`t4086`

Per ricercare informazioni relative ad un determinato modello è necessario mantenere un'altra relazione nominata `registry` per associare l'URI di un determinato documento al nome della relativa tabella. La tabella `registry` è caratterizzata dalla presenza di due campi:

- `uri`: URI del modello;
- `id`: identificativo univoco associato all'URI.

L'identificativo è ottenuto prendendo il successore del maggiore `id` presente nella tabella. Il formato usato per il nome della tabella relativa ad un oggetto specifico è `tn` dove n è la rappresentazione stringa dell'identificatore contenuto nella relazione `registry`.

uri	**id**
...	...
`cic:/Coc/Init/Logic/or.con`	4085
`cic:/Coc/Init/Peano/le_mult.con`	4086
`cic:/Algebra/Basics/Z_mult_ex.con`	4087
`cic:/Algebra/Basics/deMorgan_or_and.con`	4088
...	...

Tabella 5.3: Alcune tuple della relazione `registry`

I metadati Dublin Core, invece, sono memorizzati in tabelle con i campi `uri` (URI dell'oggetto) e `value` (valore della proprietà): ogni tabella contiene i valori di una determinata proprietà (la tabella ha lo stesso nome della proprietà) relativi a tutti gli oggetti della libreria elettronica.

Qui di seguito si descrivono brevemente i comandi la cui implementazione dipende strettamente dall'organizzazione delle relazioni del database. Il comando `relation` per ogni URI dell'insieme di risorse dato, restituisce gli elementi presenti nella tabella (associata a ciascun URI) in cui il valore del campo `prop_id` corrisponde alla proprietà data. texttpattern ricerca nella tabella `registry` tutti gli URI che corrispondono ad una delle espressioni regolari in input (restituisce delle risorse con insieme di attributi nullo). Il costrutto `attribute`, infine, restituisce il valore della proprietà per la risorsa specificata ricercandolo nella relativa tabella: al momento dell'implementazione tale costrutto è stato utilizzato unicamente per l'interrogazione delle relazioni relative ai metadati Dublin Core.

Come già detto, per aumentare il contenuto informativo dei metadati della libreria si sono aggiunte nuove proprietà alle classi RDF dello schema in modo da descrivere in modo più dettagliato gli oggetti della libreria. Poichè, come

detto più volte, l'organizzazione precedente del database era molto legata alla struttura dei file dei metadati, si sono dovute apportare delle modifiche alle relazioni del database.

Alle tabelle in cui sono memorizzati i metadati relativi agli oggetti specifici si è aggiunto il campo `depth`: contiene il valore assunto dalla proprietà omonima negli statement che descrivono gli oggetti della libreria. Tale campo assume un valore non nullo solo per gli oggetti che hanno valore del campo `position` (corrispondente al campo `context` delle tabelle del vecchio database) uguale a `MainHypothesis` o `MainConclusion`. Si sono introdotte due nuove tabelle: una per contenere i metadati relativi agli elementi `<Rel>` della rappresentazione XML degli oggetti della libreria, l'altra per i metadati riguardanti gli elementi `<Sort>`. La relazione usata per rappresentare i metadati degli elementi `<Rel>` è formata dai seguenti tre campi:

- `uri`: indica l'URI dell'oggetto contenente l'elemento `<Rel>` specificato;
- `position`: è la posizione in cui compare elemento `<Rel>` all'interno dell'oggetto;
- `depth`: indica il valore della proprietà `depth` per un determinato elemento `<Rel>`.

uri	**position**	**depth**
`cic:/Coq/Init/Datatypes/unit_rect.con`	`MainHypothesis`	0
`cic:/Coq/Init/Datatypes/unit_rect.con`	`MainConclusion`	3
`cic:/Coq/Init/Datatypes/unit_ind.con`	`MainHypothesis`	0
`cic:/Coq/Init/Datatypes/unit_ind.con`	`MainConclusion`	3
...	...	...

Tabella 5.4: Alcune tuple della relazione `refRel`

I campi della tabella dei metadati relativi agli elementi `<Sort>` sono:

- `uri`: indica l'URI dell'oggetto contenente l'elemento `<Sort>` specificato;
- `position`: è la posizione in cui compare elemento `<Sort>` all'interno dell'oggetto;

- `sort`: indica il tipo dell'elemento `<Sort>` (valore dell'omonima proprietà `sort`);
- `depth`: indica il valore della proprietà `depth` per un determinato elemento `<Sort>`.

uri	position	depth	sort
cic:/Coq/Init/projections/A.var	MainConclusion	0	Set
cic:/Coq/Init/projections/B.var	MainConclusion	Set	0
cic:/Coq/Init/unit_rect.con	MainHypothesis	Type	1
cic:/Coq/Init/unit_ind.con	MainHypothesis	Prop	1
...	...	...	...

Tabella 5.5: Alcune tuple della relazione `refSort`

5.4.2 Interprete basato su Galax

Contestualmente alla nuova versione del linguaggio Mathql è stata rilasciata anche una nuova release di Galax (*Galax-0.2-alpha*). Abbiamo ritenuto opportuno, perciò, effettuare ulteriori test delle prestazioni su Galax e aggiornare l'interprete realizzato per la precedente versione di Mathql in modo da adattarlo alle specifiche della nuova semantica di Mathql. Come già detto XQuery è un linguaggio funzionale fortemente tipato, per tale ragione abbiamo valutato anche la possibilità di implementare tutti i costrutti dell'interprete per Mathql usando direttamente XQuery, evitando, cioè, il livello intermedio realizzato in Ocaml. Inizialmente si sono testati esempi di query semplici misurandone i tempi di esecuzione. Ad esempio la seguente query Mathql:

```
select @ref1 in relation "backPointer"
ref "cic:/Algebra/NEG_anti_convert.con" attr $i
where $i = "InConclusion"
```

è stata tradotta in XQuery con:

```
declare namespace
  h="http://www.cs.unibo.it/helm/schemas/mattone.rdf#"
declare namespace
```

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

for $ref1 in document("/projects/helm/metadata/create4
    /backward/Algebra/NEG_anti_convert.con.xml")
    /rdf:RDF/h:Object/h:backPointer/h:Occurrence
  where $ref1/h:position = "InConclusion"
  return
    string($ref1/h:occurrence)
```

Un altro esempio è:

```
select @ref1 in relation "backPointer"
ref "cic:/Algebra/NEG_anti_convert.con" attr $i
where $i = "InConclusion"
union
select @ref1 in relation "backPointer"
ref "cic:/Algebra/POS_anti_convert.con" attr $i
where $i = "InConclusion"
```

in XQuery:

```
declare namespace
h="http://www.cs.unibo.it/helm/schemas/mattone.rdf#"
declare namespace
rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

for $i in (
(for $ref1 in
      document("/projects/helm/metadata/create4
      /backward/Algebra/NEG_anti_convert.con")
      /rdf:RDF/h:Object/h:backPointer/h:Occurrence
  return $ref1/h:occurrence)
union
(for $ref2 in
     document("/projects/helm/metadata/create4
     /backward/Algebra/POS_anti_convert.con")
     /rdf:RDF/h:Object/h:backPointer/h:Occurrence
```

```
 return $ref2/h:occurrence)
)
return string($i)
```

Su query semplici come quelle mostrate precedentemente i tempi di esecuzione sono comparabili a quelli ottenuti usando l'interprete che usa Postgres: in entrambi i casi le query impiegano alcuni millesimi di secondo. Nell'implementazione di query più complesse è stato necessario definire delle strutture dati opportune per rappresentare gli insiemi di risorse e i contesti utilizzati dai costrutti del linguaggio Mathql. Per manipolare tali strutture dati e per implementare i vari costrutti del linguaggio sono state definite delle apposite funzioni XQuery come descritto in dettaglio in appendice. Da un'analisi approfondita del modo di operare di Galax si è dedotto che i tempi di esecuzione delle query sono notevolmente rallentati dalla fase di parsing dei documenti XML dei metadati. Poichè gran parte delle query "significative", cioè quelle che ci si aspetta di eseguire più di frequente, accedono più volte allo stesso documento XML si è pensato di memorizzare su disco la struttura dati creata dal parser: in tal modo, infatti, se durante l'esecuzione di una query un documento viene acceduto più volte, la fase di parsing viene eseguita una sola volta. Le volte successive, invece, si usa direttamente la struttura dati, ottenuta dal parsing del documento, che era stata precedentemente salvata sul disco.

Per testare la suddetta ottimizzazione sono state usate le funzioni della libreria `Marshal` di Ocaml: per salvare la struttura dati del documento ottenuto dal parsing si è utilizzata la funzione `Marshal.to_channel()` che veniva chiamata nel modulo `stdlib/bltin.ml` all'interno della funzione `document()`. Il reperimento della struttura dati salvata in un accesso precedente allo stesso documento XML, invece, era realizzato con la funzione `Marshal.from_channel`.

Dalle prove effettuate, però, ci siamo resi conto che il salvataggio della struttura dati avveniva correttamente, ma quando tale struttura dati doveva essere riutilizzata in una chiamata successiva alla funzione `document` (sullo stesso documento XML) c'erano dei problemi. In particolare, durante la fase di parsing del documento XML, venivano create delle tabelle dinamiche referenziate dalla struttura dati restituita in output dal parser: quando la struttura dati memorizzata sul disco veniva utilizzata, senza eseguire nuovamente il parsing, i riferimenti alle tabelle dinamiche non potevano essere risolti.

5.5 Sperimentazione dell'interprete sui metadati della libreria di Helm

Per verificare l'efficienza e l'efficacia dell'implementazione fornita e per avere un valido riscontro del contenuto informativo dei "nuovi" metadati, descritti in 2.3.1, si sono studiati degli esempi "significativi" (relativamente al contesto e alle finalità del progetto Helm) di interrogazioni in modo da formulare modelli di query da generare in modo automatico.

L'idea è quella di avere un'interfaccia in cui l'utente possa fornire al generatore di query un pattern di oggetto Cic (di generalità arbitraria) in modo da ottenere in output tutti gli oggetti della libreria aventi caratteristiche in comune con il modello specificato. Ad esempio inserendo il termine `!x:nat. !P:Prop. (P x)` si desidera ottenere tutti i principi di induzione sui numeri naturali, cioè tutti quei teoremi che dato un numero naturale e una proposizione restituiscono il valore ottenuto applicando la proposizione al numero naturale.

Un altro modo di formulare la query potrebbe essere quello di inserire un pattern di tipo di dato `Cic` per ottenere l'insieme degli oggetti che sono istanze del tipo fornito: inserendo `nat -> nat`, per esempio, si possono ricercare tutte le relazioni di equivalenza sui numeri naturali.

Per costruire le query abbiamo realizzato un modulo Ocaml che partendo da due insiemi di vincoli, generati a partire dall'oggetto o dal tipo `Cic`, costruisce una query in grado di restituire l'insieme di URI degli oggetti della libreria che soddisfano tali vincoli. Gli insiemi di vincoli sono generati dal modulo Ocaml `MQueryLevels` che ispezionando l'oggetto dato in input dall'utente costruisce un insieme di vincoli "minimali" (detti anche *must*): l'insieme delle caratteristiche che tutti gli oggetti ritornati dalla query devono rispettare. L'altro insieme di vincoli, detto *only*, rappresenta un insieme di restrizioni opzionali da specificare quando si vogliono ottenere sottoinsiemi specifici dell'insieme degli oggetti che soddisfano il pattern inserito.

Dal punto di vista implementativo l'insieme dei vincoli must consiste in una lista multipla contenente tre liste: una per i vincoli sugli oggetti `Cic` in generale (le relazioni usate nelle query sono `refObj` e `backPointer`), una per i vincoli sugli elementi `Rel` (la relazione usata è `refRel`) e una per gli elementi `Sort` (relazione `refSort`). La prima è una lista di terne in cui il primo elemento è l'URI dell'oggetto, il secondo è una posizione (`MainConclusion, InConclulsion,`

...), il terzo è un elemento opzionale e rappresenta il valore della sottoproprietà `depth` di `refObj\backPointer` (come già spiegato in 2.3, tale sottoproprietà assume valori non nulli solo in corrispondenza dei valori`MainConclusion` e `MainHypothesis` della sottoproprietà `position`). L'insieme dei vincoli sui `Rel` è una lista di coppie in cui la prima componente corrisponde al valore della sottoproprietà `position` e la seconda componente (opzionale) è una `depth`. Gli elementi della terza lista, relativa ai vincoli sui `Sort`, sono delle terne in cui i primi due elementi, come per i `Rel`, sono i valori di `position` e`depth` mentre il terzo elemento contiene valori della sottoproprietà `sort`. Il tipo di dato usato per l'insieme dei vincoli only ha lo stesso formato di quello usato per i must, con la sola differenza che le tre liste che compongono la lista multipla sono opzionali (tipo di dato Ocaml `list option`).

In appendice è riportato un esempio di interrogazione in cui viene mostrato come l'interfaccia utente consente, una volta processato il pattern inserito, di specificare in vari modi i vincoli da usare nella ricerca.

Capitolo 6

Conclusioni e sviluppi futuri

Con l'introduzione di metadati per la descrizione di informazioni e con l'utilizzo del formato RDF si evita una serie di problemi di interoperabilità che costituiscono il principale ostacolo nello scambio di informazioni dell'era Internet. Tuttavia, il presente lavoro ha mostrato come l'utilizzo di soluzioni standard per la gestione di metadati è ancora in forte contrasto con requisiti prestazionali che impongono un efficiente sistema di reperimento dei dati.

In particolare ci si è resi conto che l'introduzione di costrutti generali per l'interrogazione di librerie di metadati RDF costringe a soluzioni pratiche che anche nei casi più fortunati incidono pesantemente sulle prestazioni generali del sistema. Tali considerazioni sono scaturite, in parte, dai risultati delle sperimentazioni sull'utilizzo di XQuery come meccanismo di interrogazione. Nel caso specifico della libreria elettronica di Helm, si è potuto verificare come nonostante tale linguaggio di query sia particolarmente flessibile e consenta di evitare soluzioni fortemente legate alla struttura dello schema dei metadati, i tempi di esecuzione delle query risultano ancora poco soddisfacenti per le esigenze di utilizzo nel contesto del progetto Helm: nelle interrogazioni effettuate direttamente dall'utente (di cui un esempio pratico è riportato in appendice), infatti, i tempi di risposta del sistema devono essere ridotti il più possibile. Le ottimizzazioni proposte in 5.4.2 potrebbero portare a risultati interessanti per il contesto applicativo della libreria di Helm, ma in generale si può concludere che il tempo di parsing dei documenti XML rappresenta un costo prestazionale particolarmente difficile da abbattere.

Con l'introduzione del linguaggio Mathql si è cercato di sopperire alle man-

canze del prototipo di linguaggio di interrogazione i cui punti deboli erano sati messi in luce in [Lor02]. Nella formulazione delle query da costruire con il tool di generazione automatica, però, ci si è resi conto che nell'intento di creare un linguaggio di interrogazione di metadati RDF generico, cioè adatto a schemi di metadati di origine qualsiasi, alcuni dei costrutti definiti sono risultati particolarmente inadeguati all'organizzazione del sistema di gestione di metadati di Helm. In particolare il costrutto `relation` potrebbe essere ottimizzato in modo da poter specificare i valori degli attributi delle risorse da restituire al fine di evitare inutili ricerche e scartare, già a tale livello, risultati che sarebbero comunque messi da parte in selezioni successive effettuate con l'uso di altri costrutti (tale esigenza è risultata particolarmente evidente nelle query che includono le relazioni `refRel` e `refSort`).

Nella parte finale del presente lavoro di tesi si è potuto constatare come delle leggere modifiche agli schemi RDF dei metadati , che continuano ad avere una struttura molto semplice, abbiano aumentato in modo considerevole il contenuto informativo del sistema di supporto delle metainformazioni sulla libreria. Lo sviluppo del sistema di generazione automatica delle query resta aperto a eventuali estensioni rivolte ad includere ulteriori livelli di dettaglio nella specificazione dei criteri di ricerca da parte dell'utente, nondimeno all'introduzione di nuovi modelli di interrogazione per soddisfare particolari esigenze degli utilizzatori.

Appendice A

Strutture dati dell'interprete basato su Postgres

Di seguito vengono riportate le strutture dati, in sintassi Ocaml, utilizzate nell'implementazione della versione dell'interprete per Mathql che, esegue le query usando il sistema di DBMS PostgreSQL.

```
(* Output data structures *)

(* the name of an attribute *)
type path            = string * (string list)

(* the value of an attribute *)
type value           = string list

(* an attribute *)
type attribute       = path * value

(* a group of attributes *)
type attribute_group = attribute list

(* the attributes of an URI *)
type attribute_set   = attribute_group list

(* an attributed URI *)
```

```
type resource        = string * attribute_set

(* a set of resources *)
type resource_set    = resource list

(* the query result *)
type result = resource_set


(* Input data structures *)


(* the name of a variable for a resource set *)
type svar = string

(* the name of a variable for a resource *)
type rvar = string

(* the name of a variable for an attribute value *)
type vvar = string


type inverse = bool

type assign = path * path

type refine = RefineExact
            | RefineSub
            | RefineSuper

type set_exp = SVar of svar
             | RVar of rvar
             | Ref of val_exp
             | Pattern of val_exp
             | Relation of inverse * refine * path * set_exp * assign list
```

```
             | Select of rvar * set_exp * boole_exp
             | Union of set_exp * set_exp
             | Intersect of set_exp * set_exp
             | Diff of set_exp * set_exp
             | LetSVar of svar * set_exp * set_exp
             | LetVVar of vvar * val_exp * set_exp

and boole_exp = False
              | True
              | Not of boole_exp
              | Ex of rvar list * boole_exp
              | And of boole_exp * boole_exp
              | Or of boole_exp * boole_exp
              | Sub of val_exp * val_exp
              | Meet of val_exp * val_exp
              | Eq of val_exp * val_exp

and val_exp = Const of string list
            | RefOf of set_exp
            | Record of rvar * path
            | VVar of vvar
            | Attribute of inverse * refine * path * val_exp
            | Fun of string * val_exp

type query = set_exp

(* Contexts *)

(* svars context *)
type svar_context = (MathQL.svar * MathQL.resource_set) list

(* rvars context *)
type rvar_context = (MathQL.rvar * MathQL.resource) list

(* dot-construct context *)
```

```
type group_context = (MathQL.rvar * MathQL.attribute_group) list

(* vvars context *)
type vvar_context = (MathQL.vvar * MathQL.value) list


type context = {svars: svar_context;
                rvars: rvar_context;
                groups: group_context;
                vvars: vvar_context
               }

(* updating svars context function *)
let upd_svars c s =
  {svars = s; rvars = c.rvars; groups = c.groups; vvars = c.vvars}

(* updating rvars context function *)
let upd_rvars c s =
  {svars = c.svars; rvars = s; groups = c.groups; vvars = c.vvars}

(* updating dot-construct context function *)
let upd_groups c s =
  {svars = c.svars; rvars = c.rvars; groups = s; vvars = c.vvars}

(* updating vvars context function *)
let upd_vvars c s =
  {svars = c.svars; rvars = c.rvars; groups = c.groups; vvars = s}
```

Appendice B

Strutture dati e funzioni XQuery per l'interprete

In questa appendice vengono mostrate le strutture dati e le funzioni definite per sperimentare la codifica delle query Mathql direttamente in sintassi XQuery.

```
declare namespace
h="http://www.cs.unibo.it/helm/schemas/mattone.rdf#"
declare namespace
rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

{---- general datatypes ----}

define type elem { xsd:string | type rvar
                   | type vvar | type resource
   | type attribute_group
   | type value }

define type couple { (type elem, type elem) }

define type couple_seq { type couple* }


{---- interpreter contexts ----}
```

```
define type svar_context { (type svar, type resource_set)* }

define type rvar_context { (type rvar, type resource)* }

define type group_context { (type rvar, type attribute_group)* }

define type vvar_context { (type vvar, type value)* }


{---- input data structures ----}

define type svar { xsd:string }

define type rvar { xsd:string }

define type vvar { xsd:string }

define type path { xsd:string* }

define type vvar_list { xsd:string* }

define type refine_op { xsd:string }


{---- output data structures ----}

define type resource_set { type resource* }

define type resource { xsd:string, type attribute_set }

define type attribute_set { type attribute_group* }

define type attribute_group { type myattribute* }
```

```
define type myattribute { xsd:string, type value }

define type value { xsd:string* }


{---- utility functions ----}


define function fst(type couple $c)
  returns type elem
{
subsequence($c, 1, 1)
}


define function snd(type couple $c)
  returns type elem
{
subsequence($c, 2, 2)
}


define function lassoc(type elem $k, type couple_seq $l)
  returns type elem
{
if (fst(headc($l)) = $k) then (snd(headc($l)))
else lassoc($k, (remove($l, 1)))
}


define function headc(type couple_seq $l)
  returns type couple
{
subsequence($l, 1, 2)
}
```

```
{---- interpreter functions ----}

define function ref(xsd:string* $uril)
  returns type resource_set
{
for $i in $uril
return ($i,())
}


define function
relation(type refine_op $rop, type path $p,
         type resource_set $rs, type vvar_list $vl)
returns type resource_set
{
if ($p = "refObj") then
for $r in $rs
for $ref1 in document($r)/rdf:RDF/h:Object/h:refObj/h:Occurrence
return ($ref1)
else
for $r in $rs
for $ref1 in document($r)/rdf:RDF/h:Object/h:backPointer/h:Occurrence
        return ($ref1)

}


{-- Note: $v1 and $v2 are ordered sequences --}

define function mysub(type value $v1, type value $v2)
  returns xsd:boolean
{
sequence-deep-equal(glx:intersect-values($v1, $v2), $v1)
```

```
}


define function
record(type rvar $rv, type vvar $vv,
       type group_context $gc)
returns type value
{
lassoc($vv, (lassoc($rv, $gc)))
}

define function
select(type rvar $rv, type group_context $gc,
       type rvar_context $rc, type resource_set $rso)
  returns type resource_set
{

for $i in(
 for $r in relation("", "backPointer", ref("./le.ind,1,1.xml"), "pos")
 let $rso := insert($rso, 1, $r)
 where (mysub("MainConclusion",
        record($rv, "pos", insert($gc, 1, ($rv,$r))))
 return $rso
)
return (fst($i))
}
```

Appendice C

Esempio di interrogazione della libreria Helm

Riportiamo di seguito un esempio di query costruita con il tool di generazione automatica, descritto in sez.5.5. In fig.C.1 viene mostrata la finestra iniziale dell'interfaccia utente e nel riquadro in basso a sinistra viene inserito il pattern da ricercare. Nell'esempio in questione è stato inserito il tipo `nat -> nat -> Set` per eseguire una ricerca delle relazioni sui numeri naturali. beginfigure[htbp] Selezionando `SearchPattern` dal menù `Search` si apre la finestra, mostrata in fig.C.2 che permette di specificare im modo dettagliato i vincoli, infatti si può scegliere di forzare o meno i vincoli "only" e di attivare le restrizioni su Rel, Sort e costanti. Nel nostro esempio scegliamo di forzare i vincoli "only", di restringere le `depth` delle costanti e dei Sort rispettivamente a "0" e "2" in modo da ricercare tutti i predicati binari sui numeri naturali. Cliccando su `Ok` si ottiene una lista di URI di oggetti `Cic` (figC.3). Si può eventualmente controllare se il risultato è consistente cliccando su uno degli URI della lista e contollare nella relativa finestra se il formato dell'oggetto è conforme alle nostre richieste. La query in sintassi Matql generata, in questo caso, è:

```
let $obj_positions be
"http://www.cs.unibo.it/helm/schemas/schema-helm#MainHypothesis" in
let $sort_positions be
"http://www.cs.unibo.it/helm/schemas/schema-helm#MainConclusion" in
let $sorts be
"http://www.cs.unibo.it/helm/schemas/schema-helm#Set" in
```

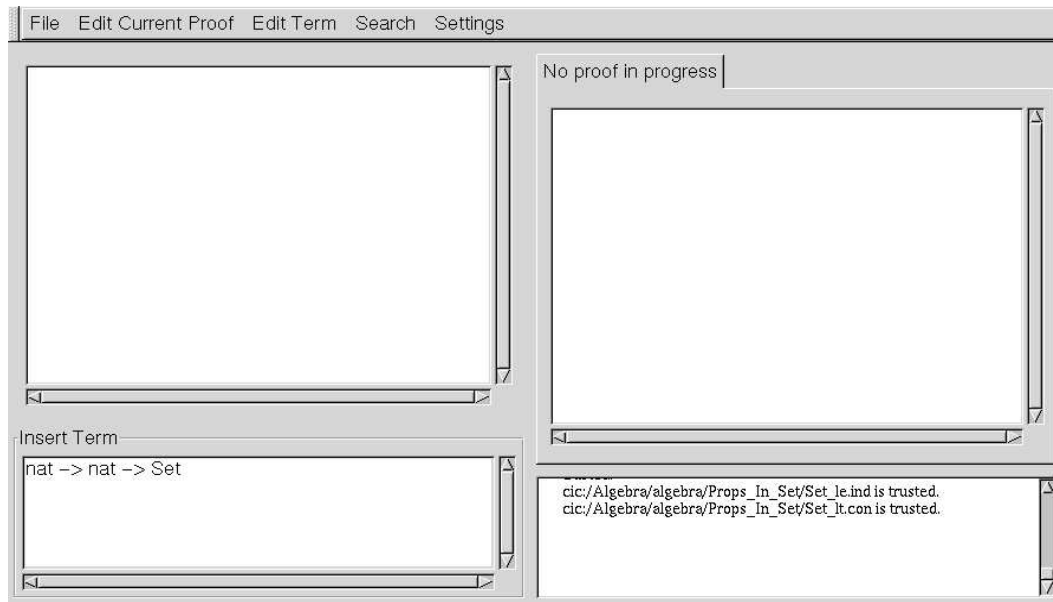

Figura C.1: Inserimento pattern

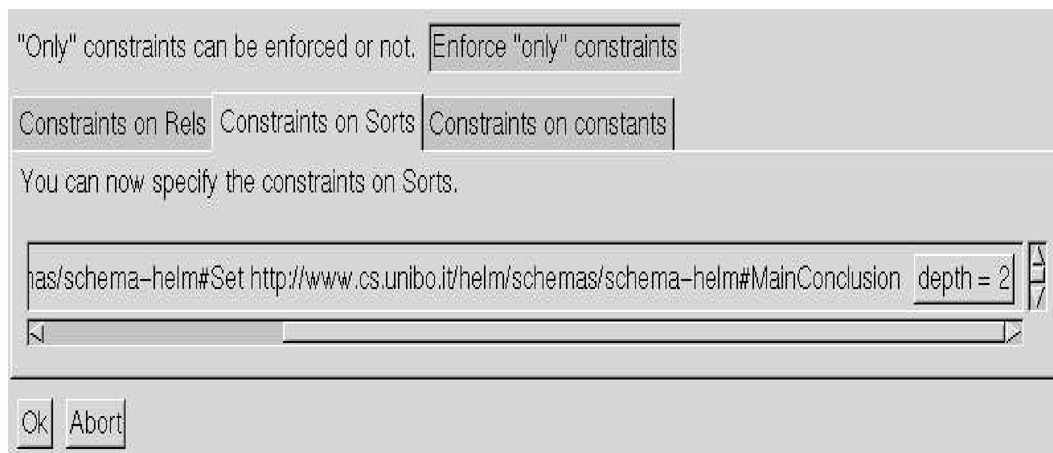

Figura C.2: Selezione vincoli

```
let $universe be "cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)" in
select @uri0 in
 (select @uri in
  relation "backPointer" "cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)"
  attr $pos <- "position"
 where ex
   ("http://www.cs.unibo.it/helm/schemas/schema-helm#MainHypothesis"
  sub
   @uri.$pos)
 intersect
 select @uri in
  relation inverse "refSort" ""
  attr $p <- "position", $d <- "depth", $s <- "sort"
 where ex
    ((
    "http://www.cs.unibo.it/helm/schemas/schema-helm#MainConclusion"
   sub
    @uri.$p)
  and
    ("http://www.cs.unibo.it/helm/schemas/schema-helm#Set"
   sub
    @uri.$s)))
where
 (((refof
  select @uri in
   relation "refObj" refof @uri0
   attr $pos <- "position"
  where ex
    ($obj_positions meet @uri.$pos)
   sub
    $universe)
  and
    (property "refRel"/"position" refof @uri0
   sub
    $rel_positions))
```

```
  and
    ((property "refSort"/"position" refof @uri0
   sub
    $sort_positions)
  and
    (property "refSort"/"sort" refof @uri0
   sub
    $sorts)))
```

Figura C.3: Risultati dell'interrogazione

In questo esempio abbiamo scelto di visualizzare l'oggetto
`cic:/Algebra/algebra/Props_In_Set/Set_lt.con` (vedi fig.C.4)

DEFINITION cic:/Algebra/algebra/Props_In_Set/Set_lt.con() OF TYPE
$\Pi\, m$: nat. $\Pi\, n$: nat. Set
AS
$\lambda\, m$: nat. $\lambda\, n$: nat.(Set_le (1, + , m) n)

Figura C.4: Visualizzazione di un oggetto

Bibliografia

[Ali] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suci. A Query Language for XML.

[APSSa] Asperti A., Padovani L., Sacerdoti Coen C., and Schena I. Content Centric Logical Environments. Short Presentation at LICS 2000.

[APSSb] Asperti A., Padovani L., Sacerdoti Coen C., and Schena I. Formal Mathematics in MathML. To be presented at MathML International Conference 2000.

[APSSc] Asperti A., Padovani L., Sacerdoti Coen C., and Schena I. Towards a Library of Formal Mathematics. Accepted at TPHTOOLS 2000.

[APSSd] Asperti A., Padovani L., Sacerdoti Coen C., and Schena I. XML, Stylesheets and the re-mathematization of formal content. Submitted to LPAR 2000.

[Com98] Communication of the ACM. Special Issue on Digital Libreries., 1998.

[dub] The Dublin Core Initiative. http://www.dublincore.org.

[GS] Guidi F. and Schena I. A Query Language for a Metadata Framework about Mathematical Resources.

[Har94] Harrison J. The QED Manifesto. In *Automated Deduction - CADE 12*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 238–251. Springer-Verlag, 1994.

[Lor02] Lordi D. Sperimentazione e sviluppo di strumenti per la gestione di metadati. Tesi universitaria, Università degli Studi di Bologna, 2002.

[OCa] The Objective Caml system release 3.06, Documentation and users manual. http://caml.inria.fr/ocaml/htmlman/.

[Posa] Galax: "The XQuery implementation for discriminating hackers" . http://db.bell-labs.com/galax/.

[Posb] Postgres: OCaml bindings for PostgreSQL. http://www.eleves.ens.fr:8080/home/frisch/soft.

[PSQ] PostgreSQL DBMS. http://www.postgresql.org.

[Ric99] Ricci A. Studio e progettazione di un modello RDF per biblioteche matematiche elettroniche. Tesi universitaria, Università degli studi di Bologna, 1999.

[Sac99] Sacerdoti C. C. Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata. Tesi universitaria, Università degli Studi di Bologna, 1999.

[W3Ca] Extensible Markup Language (XML) 1.0 (Second Edition W3C Recommendation 6 October 2000. http://www.w3.org/TR/1998/REC-xml-19980210.

[W3Cb] N-Triple. http://www.w3c.org/2001/sw/RDFCore/ntriples.

[W3Cc] Namespaces in XML W3C Recommendation 14 January 1999. http://www.w3.org/TR/REC-xml-names.

[W3Cd] PICS Label Distribution Label Syntax and Communication Protocols Version 1.1. http://www.w3c.org/TR/REC-PICS-labels.

[W3Ce] Resource Description Framework (RDF). http://www.w3c.org/TR/REC-rdf-syntax.

[W3Cf] Resource Description Framework (RDF) Schema Specification. http://www.w3c.org/TR/PR-rdf-schema.

[W3Cg] XML Linking Language (XLink) 1.0 W3C Recommendation 27 June 2001. http://www.w3.org/TR/xlink.

[W3Ch] XML Path Language (XPath) 1.0 W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath.

[W3Ci] XML Pointer Language (XPointer) 1.0 W3C Recommendation 11 September 2001. http://www.w3.org/TR/xptr.

[W3Cj] XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[W3Ck] XML Schema Part 0: Primer. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[W3Cl] XML Schema Part 2: Datatypes. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

[W3Cm] XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/query-datamodel/.

[W3Cn] XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/query-semantics/.

[W3Co] XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xquery-operators/.

[W3Cp] XQueryX, Version 1.0 . http://www.w3.org/TR/xqueryx/.

Ringraziamenti

Un ringraziamento sentito al Prof. Andrea Asperti e ai membri del progetto Helm con i quali ho lavorato nel periodo di tesi; in modo particolare ringrazio il dott. Claudio Sacerdoti Coen per la sua immensa disponibilità e per il tempo che ha dedicato a risolvere i problemi sorti nello sviluppo del software realizzato.

Ringrazio calorosamente tutti i miei amici e compagni di corso, per i loro consigli ed aiuti non solo in ambito universitario: la loro amicizia mi ha sempre confortato nei momenti difficili!

Infine ringrazio la mia famiglia, la mia ragazza e i miei compagni di appartamento per avermi pazientemente sopportato durante il mio lungo periodo di tesi e per la capaciatà di condividere ogni mia scelta e decisone.