Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Materia di Tesi: Informatica Teorica

Web services per il supporto alla dimostrazione interattiva

Tesi di Laurea di:
Stefano Zacchiroli

Relatore:
Chiar.mo Prof. Andrea Asperti

Correlatore:
Dott. Claudio Sacerdoti Coen

III 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

Web services per il supporto alla dimostrazione interattiva

Tesi di Laurea di:
Stefano Zacchiroli

Relatore:
Chiar.mo Prof. Andrea Asperti
Correlatore:
Dott. Claudio Sacerdoti Coen

Parole chiave:
Web Services; Proof Assistant; XML; XSLT;
Mathematical Knowledge Management

III Sessione
Anno Accademico 2001-2002

A chi si è preoccupato più di quanto avrebbe dovuto,
a chi si è preoccupato enormemente meno di quanto avrebbe dovuto.

A Gaia, Gaspa, Iolanda, Rita e Silvia.

Ai miei genitori.

Indice

1 Introduzione **1**
1.1 Struttura della tesi . . . 3
1.2 Servizi Web . . . 4
1.2.1 Il Web semantico . . . 5
1.2.2 Definizione delle interfacce: WSDL . . . 6
1.2.3 Standardizzazione dei servizi web . . . 7
1.2.4 Interfacce web a servizi già esistenti . . . 8
1.3 Il progetto HELM . . . 9
1.3.1 Architettura . . . 10
1.3.2 Componenti . . . 11
1.3.3 Interazioni . . . 15
1.3.4 I client . . . 16

2 *OCamlHTTP*: server HTTP in OCaml **27**
2.1 Il protocollo HTTP . . . 28
2.1.1 Generalità . . . 28
2.1.2 Formato dei messaggi . . . 30
2.1.3 Identificazione della risorsa richiesta . . . 31
2.1.4 Metodi . . . 32
2.1.5 Codici di stato . . . 34
2.1.6 Header . . . 34
2.1.7 Connessioni . . . 35
2.2 Architettura . . . 36
2.2.1 Inizializzazione di demoni HTTP . . . 38
2.2.2 Modalità di gestione dei client . . . 43
2.2.3 Formalizzazione dei messaggi HTTP . . . 45
2.3 Invio di risposte ai client . . . 50
2.4 URI escaping . . . 54
2.5 Descrizione dei moduli . . . 55

2.6 Esempi di utilizzo 56
2.7 Implementazione 56

3 *HTTP Getter*: gestione della base documentaria 61
3.1 Modello di distribuzione 62
3.1.1 APT – Advanced Package Tool 63
3.1.2 HELM getter 64
3.2 Obiettivi progettuali 65
3.3 Formato dei nomi logici (URN) 65
3.3.1 Schema *object:* 66
3.3.2 Schema *theory:* 67
3.3.3 Schema *rdf:* 67
3.4 Interfaccia 68
3.4.1 Metodo *getxml* 68
3.4.2 Metodo *getdtd* 69
3.4.3 Metodo *getxslt* 69
3.4.4 Metodo *resolve* 70
3.4.5 Metodo *register* 70
3.4.6 Metodo *update* 71
3.4.7 Metodo *getalluris* 71
3.4.8 Metodo *getallrdfuris* 71
3.4.9 Metodo *ls* 72
3.4.10 Metodo *getempty* 73
3.4.11 Metodo *help* 73
3.5 Architettura 74
3.5.1 Base documentaria 74
3.5.2 Struttura interna 74
3.5.3 Descrizione dei moduli 76
3.6 Confronto con le precedenti implementazioni 78
3.7 Implementazione 79

4 *UWOBO*: processore di *catene* XSLT 81
4.1 Interfaccia 82
4.1.1 Metodo *add* 82
4.1.2 Metodo *remove* 82
4.1.3 Metodo *reload* 83
4.1.4 Metodo *list* 83
4.1.5 Metodo *apply* 83
4.1.6 Metodo *help* 84

4.2 Architettura 85
4.2.1 Descrizione dei moduli 86
4.3 Gestione dei fogli di stile 87
4.4 Catene di fogli di stile 89
4.4.1 Fogli di stile *parametrici* 90
4.4.2 Proprietà di output 91
4.5 Problemi di sviluppo 93
4.6 Confronto con le precedenti implementazioni 93
4.7 Implementazione 95

5 *drawGraph*: generazione di grafi di dipendenza 97
5.1 Tipologie di grafi 97
5.2 Architettura 99
5.2.1 Graphviz 101
5.3 Interfaccia di *drawGraph* 102
5.3.1 Metodo *draw* 102
5.3.2 Metodo *get_gif* 103
5.3.3 Metodo *help* 103
5.4 Interfaccia di *uriSetQueue* 103
5.4.1 Metodo *set_uri_set_size* 103
5.4.2 Metodo *add_if_not_in* 104
5.4.3 Metodo *get_next* 104
5.4.4 Metodo *is_overflowed* 105
5.4.5 Metodo *reset_to_empty* 105
5.4.6 Metodo *help* 105
5.5 Implementazione 106

6 *searchEngine*: ricerche all'interno della libreria 107
6.1 Architettura 109
6.1.1 Disambiguazione 109
6.2 Interfaccia 111
6.2.1 Metodo *execute* 111
6.2.2 Metodo *locate* 112
6.2.3 Metodo *searchPattern* 112
6.2.4 Metodo *getPage* 113
6.3 Implementazione 113

7 *HBugs*: supporto alla dimostrazione interattiva 115
7.1 Terminologia . . . 116
7.2 Architettura . . . 117
7.2.1 Una sessione di HBugs . . . 118
7.2.2 HBugs common . . . 119
7.2.3 HBugs broker . . . 121
7.2.4 HBugs tutor . . . 124
7.2.5 gTopLevel (HBugs client) . . . 129
7.3 Messaggi . . . 130
7.4 Serializzazione dello stato . . . 131
7.4.1 Stato di gTopLevel . . . 131
7.4.2 Serializzazione . . . 133
7.5 Serializzazione dei suggerimenti . . . 134
7.6 Implementazione . . . 136

8 Conclusioni 143
8.1 Componenti implementate . . . 144
8.1.1 OCamlHTTP . . . 144
8.1.2 HTTP Getter . . . 144
8.1.3 UWOBO . . . 146
8.1.4 drawGraph . . . 147
8.1.5 searchEngine . . . 148
8.1.6 HBugs . . . 149
8.2 Sviluppi futuri . . . 150

A *WST*: tester per servizi web 155
A.1 Implementazione . . . 157

Bibliografia 158

Elenco delle figure

1.1 HELM: architettura logica 11
1.2 HELM: dipendenze HTTP 15
1.3 HELM web site: indice 17
1.4 HELM web site: una definizione 18
1.5 HELM web site: proof checking 18
1.6 HELM web site: metadata 19
1.7 HELM web site: grafo di dipendenza 20
1.8 HELM web site: pipeline HTTP 20
1.9 HELM web site: trasformazioni XSLT 22
1.10 gToplevel: screenshot 24

2.1 HTTP: esempi di messaggi 30
2.2 Tipologie di connessioni HTTP 37
2.3 Vita di un server HTTP 38
2.4 Messaggi HTTP - Ereditarietà 45
2.5 Dipendenza tra i moduli di OCamlHTTP 57
2.6 Esempio di utilizzo di OCamlHTTP: approccio funzionale . . . 58
2.7 Esempio di utilizzo di OCamlHTTP: approccio imperativo . . . 59

3.1 Modello di distribuzione di APT 63
3.2 Getter, metodo *ls*: formati di output 73
3.3 Getter, metodo *ls*: DTD del formato di output XML 74
3.4 *getter*: architettura 75
3.5 *getter*: dipendenza dei moduli 76

4.1 UWOBO: architettura 85
4.2 UWOBO: dipendenza dei moduli 87

5.1 Principio di induzione su liste e accesso ai relativi metadati . . . 98
5.2 Principio di induzione su liste: dipendenze *forward* 99

5.3 Principio di induzione su liste: dipendenze *backward* 99
5.4 Generatore di grafi di dipendenza: architettura 100
5.5 Graphviz: descrizione testuale di un grafo orientato 102

7.1 HBugs: architettura 117
7.2 Hbugs_common: dipendenza dei moduli 120
7.3 Hbugs_broker: interfacce 121
7.4 Hbugs_broker: architettura 122
7.5 Hbugs_broker: dipendenza dei moduli 124
7.6 Hbugs_tutor: architettura 125
7.7 HBugs: input del funtore di generazione dei tutor 127
7.8 HBugs: implementazione del tutor Ring 128
7.9 HBugs: descrizione XML di un tutor 128
7.10 Hbugs_client: architettura 129
7.11 gTopLevel: stato 133

A.1 WST (Web Service Tester) 157

Elenco delle tabelle

1.1 HELM: componenti 12

4.1 UWOBO: proprietà di output 92

7.1 HBugs: messaggi client → broker 132
7.2 HBugs: messaggi tutor → broker 138
7.3 HBugs: messaggi broker → client 139
7.4 HBugs: messaggi broker → tutor 140
7.5 HBugs: messaggi generici 141

Capitolo 1

Introduzione

Dalla sua nascita nel 1989 ad oggi, il *World Wide Web* si è affermato come il servizio più utilizzato all'interno della rete *Internet*. La popolarità raggiunta da questo servizio e la penetrazione sul mercato dei prodotti software che permettono di accedervi hanno contribuito in maniera considerevole alla ulteriore diffusione della rete Internet e al proliferare dei punti di accesso ad essa.

La maggiore accessibilità alla rete Internet si è riflessa nel campo delle applicazioni software non solo nello sviluppo e nella nascita di nuove tipologie di software strettamente correlate ad Internet stessa, ma anche cambiando sostanzialmente il modo di strutturare sistemi software complessi.

Tipologie di software che fino a pochi anni fa erano state interamente pensate per essere monolitiche e fruibili da un singolo computer hanno subito negli ultimi anni una rivoluzione strutturale che le ha avvicinate alle architetture *client/server* tipiche del mondo delle reti e, più in generale, di Internet. Per citare solo alcuni esempi possiamo pensare al mondo dei DBMS, ai sistemi di controllo di versione, all'intrattenimento ecc.

Storicamente le applicazioni client/server sono però state caratterizzate da problematiche di compatibilità fra architetture diverse (ancora oggi capita di scontrarsi con *bug* in software di dimensioni ragguardevoli dipendenti dalla *endianness*[1] delle architetture), sistemi operativi diversi, versioni diverse di uno stesso programma ecc. Nella maggior parte dei casi non esiste una soluzione definitiva a questi problemi di compatibilità, essi vengono quindi risolti standardizzando un linguaggio, un formato di dato, una semantica.

Gli sforzi di standardizzazione del *W3C* hanno dato un grosso contributo alla soluzione di problemi di standardizzazione. In particolare il metalinguaggio XML, standardizzato dal W3C nel 1998, permette di definire altri linguaggi

[1] formato di rappresentazione in memoria degli interi che occupano più di un byte

occupandosi a priori di annosi problemi di incompatibilità quali, ad esempio, la codifica dei caratteri.

Avendo a disposizione questi strumenti e il World Wide Web (già regolamentato da standard quali il protocollo *HTTP*) la migrazione di architetture client/server verso il mondo del web sta sempre più prendendo piede. Il Web sta infatti diventando sempre più strutturato e ricco di informazioni fruibili non solo da un utente che visualizza una pagina utilizzando un *browser*, ma anche da agenti automatici che siano in grado di interpretare le informazioni disponibili e di elaborarle.

Lo sviluppo del Web ha inoltre di fatto imposto un nuovo modello di architettura software: quello delle applicazioni *document centric* (o *content centric*). Questo modello si contrappone al modello *application centric*, ampiamento diffuso fino alla prima metà degli anni '90, che vedeva nella applicazione lo strumento principe di accesso alle informazioni.

Seguendo questo principio, le informazioni non sono importanti di per sè, ma solo in quanto esistono una o più applicazioni in grado di elaborarle. Questa filosofia ha portato ad un proliferare di formati proprietari e ha ulteriormente peggiorato gli scenari di incompatibilità già analizzati.

Nel modello document centric, che risolve gran parte dei problemi del modello application centric, l'enfasi è invece posta sull'informazione che deve di conseguenza godere di buone proprietà quali: la standardizzazione del formato in cui è espressa, l'indipendenza dalle applicazioni che la utilizzano, la buona strutturazione. Tutte proprietà implementabili con successo utilizzando XML.

Il modello document centric permette di sviluppare applicazioni software più modulari. Unico requisito di queste applicazioni è la condivisione della macro struttura delle informazioni.

Le tipologie di applicazioni che maggiormente hanno da beneficiare dai modelli client/server e document centric sono le librerie di conoscenza strutturata. Strutturando la base di dati in un insieme di documenti XML diventa infatti possibile, a partire dai medesimi documenti, presentarli in HTML in modo che siano fruibili dai "normali" utilizzatori di browser, ma anche scrivere motori di ricerca ad hoc per la base documentaria, raccogliere annotazioni esterne relative a particolari parti di un documento, studiare sistemi di hyperlink ad hoc ecc.

Nell'ambito della matematica formale e dei sistemi di supporto alla dimostrazione interattiva (*proof assistant*) sono disponibili basi di dati contenenti formalizzazioni di dimostrazioni di teoremi matematici in modo da permettere il loro riutilizzo nel tentativo (interattivo per quanto riguarda i proof assistant o automatico per quanto riguarda i *theorem proover*) di dimostrare nuovi risultati

o nuove formalizzazioni di risultati già noti.

L'utilizzo delle tecnologie sopra descritte abbinato a queste librerie di conoscenza matematica formalizzata amplia le possibilità dei tipici proof assistant permettendo, fra le tante possibilità:

- la generazione automatica di una rappresentazione comprensibile a chiunque conosca il linguaggio universale della matematica di ogni dimostrazione presente all'interno della libreria
- la pubblicazione su Web di queste rappresentazioni
- la ricerca di tutti i teoremi applicabili per concludere un passaggio di una dimostrazione
- la generazione di grafi di dipendenza di dimostrazioni, lemmi, assiomi, definizioni, ecc.
- la divisione del processo di dimostrazione tra più agenti che lavorano concorrentemente

Ognuno dei servizi sopra descritti può inoltre essere reso fruibile dalla comunità di scienziati e ricercatori interessati creando un apposito servizio web che sia utilizzabile accedendovi utilizzando la rete Internet e il protocollo HTTP secondo una interfaccia nota. La descrizione di tale interfaccia deve comprendere un punto di accesso al servizio (un *URL*) ed una descrizione del formato dei parametri di input e output del servizio.

In questa tesi analizzeremo lo sviluppo di una serie di servizi web utilizzati nell'ambito del progetto HELM e i vantaggi offerti da questo approccio nell'ambito delle librerie di conoscenza matematica formalizzata.

1.1 Struttura della tesi

La presente tesi è divisa in cinque parti.

I (cap. 1) consiste di questa introduzione

II (cap. 2) descrive la libreria implementata per lo sviluppo di servizi web basati sul protocollo HTTP

III (capp. 3 – 7) descrive le componenti di HELM implementate nell'ambito di questa tesi:

Getter (cap. 3) servizio web responsabile della gestione della base documentaria

UWOBO (cap. 4) servizio web responsabile della applicazione di catene di fogli di stile XSLT

drawGraph (cap. 5) servizio web responsabile della generazione di grafi di dipendenza tra oggetti contenuti all'interno della base documentaria

searchEngine (cap. 6) insieme di servizi web che permettono di effettuare ricerche all'interno della base documentaria

HBugs (cap. 7) infrastruttura, basata su diversi servizi web, in grado di fornire aiuti all'utente durante il processo di dimostrazione interattiva con il proof assistant *gTopLevel*

IV (cap. 8) riporta valutazioni conclusive a riguardo del lavoro svolto

V (app. A) descrive una componente software sviluppata a supporto del lavoro svolto in questa tesi

Nella rimanente parte di questa tesi si assume una buona conoscenza della famiglia di linguaggi derivanti da XML (in particolare XSLT e MathML) e nel campo della logica formale e dei sistemi di supporto alla dimostrazione interattiva.

1.2 Servizi Web

> *A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility*
>
> Vannevar Bush, 1945

Il fatto che il World Wide Web venga sempre più frequentemente utilizzato come mezzo di comunicazione tra applicazioni ha incentivato le iniziative del W3C[2] atte alla sua standardizzazione.

[2]il *World Wide Web Consortium*, ente creato nell'Ottobre del 1994 per promuovere standard correlati al mondo del World Wide Web. Per maggiori informazioni: `http://www.w3.org`

Un **servizio web**[3] è un sistema software identificato da una URI ([RFCc]) le cui interfacce e rappresentazioni nei protocolli sottostanti sono definite utilizzando il meta linguaggio XML. Le sue interfacce possono essere rintracciate da altri sistemi software che possono poi interagire con il servizio web secondo le modalità definite nelle stesse. Le interazioni con un servizio web avvengono mediante scambio di messaggi XML utilizzando uno o più protocolli della rete Internet.

In questa sezione analizzeremo il concetto di web semantico e i suoi rapporti con i servizi web, vedremo brevemente il linguaggio proposto dal W3C per definire le interfacce dei servizi web, infine proporremo un approccio interessante per la generazione automatica di interfacce web a applicazioni già esistenti.

1.2.1 Il Web semantico

Lo stato dell'arte dei siti web su internet è piuttosto desolante dal punto di vista della fruibilità delle informazioni disponibili. Le motivazioni di tale scarsa fruibilità sono da ricercarsi per larga parte nella scarsa strutturazione delle informazioni disponibili.

Queste informazioni sono tipicamente codificate in linguaggi detti di *markup* che sono in grado di codificare informazioni testuali e altre meta informazioni (il markup) che esprimono proprietà sulle informazioni stesse.

Possiamo distinguere diverse tipologie di markup, nel nostro ambito le due tipologie[4] che ci interessano sono quelle del *markup presentazionale* e del *markup strutturale* (o descrittivo).

Lo scopo del markup presentazionale è quello di descrivere la resa grafica di una quantità di informazioni (ad es. informazioni tipografiche quali font, dimensione, colore del testo).

Lo scopo del markup strutturale è invece quello di stabilire il ruolo all'interno del documento di una quantità di informazioni (ad es. quale parte del testo rappresenta il nome di un libro, quale la sua casa editrice, quale il suo ISBN ecc.).

La quasi totalità del markup utilizzato nella stragrande maggioranza dei siti web è markup presentazionale. Pagine web di questo tipo sono "piatte" per quanto riguarda la strutturazione dell'informazione, richiedono una presenza

[3] secondo il glossario dei servizi web del W3C ([W3Ce])

[4] altre tipologie "interessanti" di markup sono: puntuazionale, procedurale, referenziale e metamarkup

umana che sia in grado di comprendere il contenuto informativo della pagina stessa.

L'idea del *web semantico*, nonché lo scopo della corrispettiva attività intrapresa dal W3C, è quella di promuovere la strutturazione delle informazioni presentate attraverso il WWW in modo che esse possano essere "comprese" anche da applicazioni software. Si passerebbe in questo modo da informazioni *leggibili* da elaboratori a informazioni *comprensibili* da elaboratori.

Gli strumenti a disposizione per la realizzazione di questo "ideale" (come lo definisce lo stesso Tim Berners Lee) sono già disponibili:, i due principali sono certamente XML per la strutturazione dei dati e RDF per la descrizione di meta informazioni.

I servizi web si integrano nell'idea del web semantico ponendosi come successori delle attuali applicazioni basate sul WWW. La maggior parte di tali applicazioni sono attualmente basate su semplici form HTML che ricevono in input dati "piatti" (ovvero non strutturati forniti da un utente) e forniscono in output dati altrettanto "piatti" (tipicamente pagine HTML mal strutturate). Con il passaggio ai servizi web sia l'input che l'output che la descrizione stessa del servizio diventano informazioni strutturate (in XML) comprensibili ad una applicazione software che disponga di un parser XML.

Diventa così possibile immaginare applicazioni che, ad esempio, effettuino per noi ricerche su una serie di motori di ricerca da noi scelti, eseguano qualche euristica sui risultati ottenuti e ci mostrino i risultati ordinati secondo un criterio sempre di nostra scelta. Rientrando nell'ambito delle applicazioni matematiche possiamo immaginare motori di ricerca ad hoc che esplorino le librerie matematiche formalizzate in diversi sistemi logici e permettano il confronto tra le diverse formalizzazioni di uno stesso teorema.

1.2.2 Definizione delle interfacce: WSDL

Uno degli anelli fondamentali per l'integrazione dei servizi web nell'idea del web semantico è la descrivibilità dei servizi disponibili.

Affinché applicazioni software possano usufruire di servizi web autonomamente è infatti senza dubbio necessario che i servizi siano descritti in un formato strutturato. A questo scopo è stato studiato ed è in corso di sviluppo il linguaggio WSDL ([W3Cd]).

L'idea fondante di questo linguaggio è quella di fornire uno strumento utilizzando il quale sia possibile descrivere in un formato comprensibile ad una applicazione le caratteristiche fondamentali di un servizio web:

- i punti di accesso del servizio (specificati da uno o più URI)
- il formato dei messaggi che possono essere scambiati tra client e server
- i sotto servizi disponibili
- le operazioni disponibili per ognuno dei sotto servizi
- i protocolli di comunicazione utilizzabili per accedere al servizio
- il comportamento del servizio in caso di fallimento

Purtroppo WSDL è ancora piuttosto immaturo e non è ancora stato standardizzato (attualmente esiste solamente una *Working Draft* del W3C).

Per una descrizione più dettagliata di WSDL si veda [W3Cd].

1.2.3 Standardizzazione dei servizi web

Gli sforzi di standardizzazione dei servizi web da parte del W3C sono stati delegati a quattro distinti *Working Group*:

1. Web Services Architecture Working Group
2. XML Protocol Working Group
3. Web Services Description Working Group
4. Web Services Choreography Working Group

Il primo (**Web Services Architecture**[5]) è il gruppo responsabile della realizzazione dei documenti "architetturali" dei web services, in pratica stabilisce come le varie componenti dei servizi web debbano collaborare tra loro.

Il secondo (**XML Protocol**[6]) è incaricato di sviluppare una infrastruttura che permetta la comunicazione tra applicazioni mediante scambio di messaggi XML e di definire un modello di comunicazione tra client e server di un servizio web. È il gruppo che si occupa anche della standardizzazione di SOAP (Simple Object Acces Protocol) ([W3Cb], [W3Cc]).

Il terzo (**Web Services Description**[7]) si occupa della realizzazione di un linguaggio XML che permetta di descrivere le interfacce di servizi web. È il gruppo che si occupa della standardizzazione di WSDL.

[5] `http://www.w3.org/2002/ws/arch/`
[6] `http://www.w3.org/2000/xp/Group/`
[7] `http://www.w3.org/2002/ws/desc/`

Il quarto gruppo (**Web Services Choreography**[8]) si occupa della formalizzazione di un linguaggio che permetta di "comporre" servizi web (inviare ad un servizio web un certo input e raccoglierne l'output per inviarlo ad un secondo servizio web ad esempio) e di descrivere le relazioni tra servizi web (ad esempio nel caso in cui un servizio web sia stato realizzato utilizzandone altri).

Esiste inoltre un quinto gruppo super partes: il **Web Services Coordination**[9] group il cui scopo è quello di coordinare il lavoro degli altri quattro gruppi.

1.2.4 Interfacce web a servizi già esistenti

Un aspetto interessante e ancora poco esplorato dei servizi web è la possibilità di generare automaticamente servizi web che interfaccino servizi software già esistenti. Ne riportiamo di seguito una breve intuizione.

La formalizzazione della descrizione delle interfacce di servizi web del linguaggio WSDL permette di utilizzare un qualunque sistema di tipi per definire il formato dei messaggi scambiati con un certo servizio web. Di fatto però, per garantire maggiore accessibilità ad un servizio, viene fortemente consigliato l'utilizzo di XML Schema ([W3Cf], [W3Cg], [W3Ch]) come sistema di tipi.

Avendo a disposizione l'implementazione di un qualche servizio software è possibile generare in maniera automatica un server per un qualche servizio Internet che implementi il servizio web associato.

Si pensi ad esempio che il generico "servizio" sia una semplice funzione di libreria f che riceva in input n valori ognuno di essi avente tipo t_i con $i = 1 \dots n$ e che restituisca in output un valore di tipo t_{out}.

Supponendo che il sistema di tipi del nostro linguaggio (nel nostro caso OCaml) sia sufficientemente ricco, possiamo stabilire una regola di conversione che mappi ogni tipo primitivo di OCaml in un tipo primitivo XML Schema (regola base) ed ogni tipo concreto OCaml su tipi complessi di XML Schema che descrivono documenti XML (regola induttiva).

Basandosi su queste assunzioni una interfaccia di modulo OCaml (contenuta all'interno dei file `.mli`) è un buon punto di partenza, anche se ancora non sufficiente, per la generazione automatica di una descrizione di un servizio web. Non è ancora sufficiente in quanto la descrizione WSDL dei servizi web richiede altre informazioni, quali ad esempio il raggruppamento delle funzionalità del servizio web in portType, che non sono rappresentabili utilizzando la sintassi di

[8] `http://www.w3.org/2002/ws/chor/`
[9] `http://www.w3.org/2002/ws/cg/`

OCaml. Queste informazioni mancanti possono però essere aggiunte utilizzando tecniche di *literate programming* ([Knu92], [FM03]).

Utilizzando questo approccio è quindi possibile generare descrizioni WSDL di servizi web automaticamente a partire da descrizioni di moduli OCaml opportunamente commentate. Avendo a disposizione anche l'implementazione degli stessi moduli è inoltre possibile generare automaticamente le componenti software che implementano il servizio web.

Se infatti scegliamo come binding per servizi web una implementazione HTTP dello stesso basata sul metodo GET, ci accorgiamo che il codice che implementa il server web, parsing dei parametri e pretty printing delle risposte è immutabile. Possiamo quindi immaginare una architettura nella quale esiste una implementazione comune di queste funzionalità e che invoca il modulo contenente la reale implementazione delle funzionalità del servizio web. Questo modulo lavorerà su semplici valori OCaml e potrà essere totalmente ignaro della parte web.

L'aggiunta di altri binding, quali ad esempio SOAP ([W3Cb], [W3Cc]), può essere effettuata modificando solamente il codice del generatore automatico senza modificare le funzionalità dei singoli servizi web.

Questa intuizione è ancora oggetto di studio e verrà sviluppata come estensione della presente tesi.

1.3 Il progetto HELM

Il progetto HELM è un progetto a lungo termine sviluppato presso l'Università di Bologna dal prof. Andrea Asperti e dal suo gruppo di ricerca. Lo scopo principale di HELM (Hypertextual Electronic Library of Mathematics) è quello di sviluppare una infrastruttura tecnologica per la creazione e la gestione di una libreria ipertestuale di conoscenza matematica formalizzata. L'implementazione della libreria deve poter contenere documenti matematici provenienti dalle librerie dei sistemi già esistenti per la gestione di conoscenza matematica.

Riportiamo in questo sezione una breve descrizione dell'architettura di HELM, per una descrizione più dettagliata si vedano [Sac00], [APS+], [APSSb], [APSSa].

Gli obiettivi progettuali di HELM sono molteplici:

standardizzazione tutti i documenti della libreria devono essere codificati in formati standard definiti utilizzando il meta linguaggio di markup XML. L'elaborazione di tali documenti deve essere, quando possibile, implemen-

tata utilizzando linguaggi standard quali, ad esempio, il linguaggio di trasformazioni XSLT

distribuzione la libreria deve essere distribuita e poter sfruttare le potenzialità dei moderni sistemi distribuiti quali replicazione e distribuzione di carico. Per implementare al meglio questi requisiti la libreria deve disporre di uno schema di nomi logici per i documenti della base documentaria che sia indipendente dalla locazione fisica ed implementare i necessari meccanismi di risoluzione

collaborazione i requisiti posti per poter aggiungere un documento alla base documentaria devono essere minimi, attualmente l'unico requisito imposto è la disponibilità di uno spazio HTTP, FTP o NFS. La libreria deve inoltre essere accessibile utilizzando strumenti software semplici e diffusi quali browser

modularità le componenti software sviluppate per l'utilizzo della libreria devono essere più modulari possibili e facilmente combinabili tra loro, è opportuno inoltre che ogni componente sia accompagnata da una interfaccia che descriva le sue modalità di utilizzo

compatibilità il progetto HELM non ha scopi fondazionali e di conseguenza non definisce *il* formalismo logico utilizzato all'interno della libreria. Deve essere possibile supportare qualsiasi formalismo logico implementando eventualmente opportuni strumenti di esportazione da quel formalismo in XML

1.3.1 Architettura

L'architettura logica di HELM ed il suo rapporto con i sistemi software esistenti che gestiscono conoscenza matematica formalizzata è schematizzabile come in fig. 1.1.

Possiamo distinguere quattro diversi livelli nell'architettura di HELM.

Il **livello 0** è quello dei sistemi software già esistenti basati sui diversi logical framework. Le librerie di conoscenza matematica già esistenti per ognuno di essi possono essere codificate in un formato XML è rese parte della libreria di HELM. Sono già stati realizzati moduli software per l'esportazione in XML delle librerie matematiche dei proof assistant Coq e NuPRL, è auspicabile la realizzazione di moduli analoghi per gli altri sistemi già esistenti.

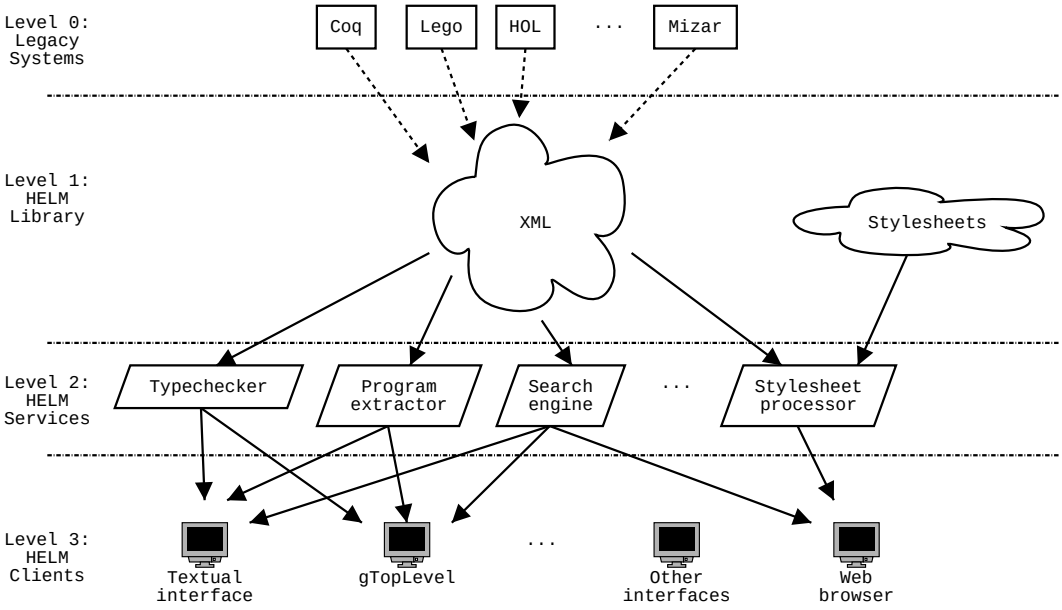


Figura 1.1: HELM: architettura logica

Il **livello 1** consiste della libreria matematica (distribuita) di HELM e dell'infrastruttura necessaria per accedere ai documenti in essa presenti utilizzando uno schema di nomi logici. A questo livello appartengono inoltre le librerie di fogli di stile XSLT descriventi trasformazioni sui documenti della libreria e i DTD delle diverse tipologie di documenti XML utilizzate.

Il **livello 2** consiste di un insieme di servizi software che utilizzano la libreria per fornire diverse funzionalità quali: type checking, resa delle dimostrazioni, motori di ricerca, estrazione di codice, ecc.

Il **livello 3** è formato dai diversi client del progetto HELM che utilizzano le funzionalità del livello precedente fornendo una interfaccia all'utente finale. Attualmente in questo livello sono presenti il sito web `http://helm.cs.unibo.it` ed il proof assistant *gTopLevel*.

1.3.2 Componenti

Dato l'obiettivo di modularità del progetto HELM la sua struttura è costituita da molte componenti che interagiscono tra loro.

Le diverse componenti sono accessibili utilizzando diverse tipologie di interfacce:

- HTTP

- API OCaml
- script eseguibili in modalità batch
- query SQL

L'accesso HTTP rispecchia la natura modulare di HELM e permette di utilizzarne i servizi in combinazione con molte delle moderne tecnologie standardizzate dal W3C che si basano sempre più massicciamente su questo protocollo. L'accesso dal linguaggio OCaml permette di realizzare componenti software che sfruttino le potenzialità della libreria di HELM quali il proof assistant gTopLevel. L'accesso via script eseguibili in modalità batch risulta necessario per la generazione di dati che non cambiano frequentemente nel tempo. L'accesso via query SQL viene utilizzato per accedere a dati che richiedano prestazioni e strutturazione tipica dei DBMS.

Le componenti principali del progetto, divise per livello di appartenenza, sono descritti in tab. 1.1.

Livello 0	server che ospitano i documenti XML, modulo di esportazione dal proof assistant Coq, modulo di esportazione dal proof assistant NuPRL
Livello 1	getter (sez. 3), generatore di metadati, database per la gestione dei metadati
Livello 2	motore di ricerca (sez. 6), type checker, UWOBO (sez. 4), HBugs (sez. 7), generatore di grafi di dipendenza (sez. 5)
Livello 3	sito web `http://helm.cs.unibo.it`, gTopLevel

Tabella 1.1: HELM: componenti

La libreria di HELM, per obiettivo progettuale, non è centralizzata, ma distribuita. Non esiste quindi un unico locazione fisica che ospiti l'intera libreria, ma più **server** ognuno dei quali ospita una parte della libreria. Attualmente HELM supporta come server ospitanti documenti matematici server HTTP, server FTP e server NFS.

I **moduli di esportazione** attualmente implementati permettono di esportare le librerie matematiche dei proof assistant Coq e NuPRL in formati XML (dei quali sono stati definiti i rispettivi DTD). L'esportazione delle librerie viene tipicamente eseguita in modalità *batch*[10] generando un albero di file XML

[10]le librerie dei sistemi finora supportati non cambiano così frequentemente da consigliare un approccio diverso

all'interno di un filesystem, questo albero viene poi abbinato ad un file di indice (generabile automaticamente durante il processo di esportazione) che descriva i documenti contenuti. L'albero di file XML ed il file di indice vengono poi pubblicati utilizzando un server delle tipologie supportate da HELM (HTTP, FTP, NFS). I moduli di esportazione non dispongono di interfacce HTTP o API per OCaml in quanto non necessitano di nulla del genere, sono tipicamente script o programmi da utilizzare *una tantum* per l'esportazione.

Il **getter** è la componente che fornisce accesso alla libreria per conto delle componenti dei livelli superiori e per il generatore di metadati. Implementa lo schema di nomi logici che astrae sopra la locazione fisica dei documenti e implementa i meccanismi di risoluzione di tali nomi. Il getter è pensato per essere fisicamente *vicino* all'utente finale ed è configurabile selezionando quali server utilizzare per la costituzione della *vista* sulla libreria matematica dell'utente. Il getter fornisce entrambe le interfacce tipiche delle componenti di HELM: HTTP e API OCaml.

Abbinato al getter vi sono il **generatore di metadati** ed il database utilizzato per contenerli. Il processo di generazione di metadati RDF per la libreria è infatti (fortunatamente) in larga parte automatizzabile, in particolare attualmente è automatizzata la generazione dei seguenti metadati: termini da cui dipende un termine della libreria (*forward*), termini che utilizzano un dato termine della libreria (*backward*), costruttori, tipi induttivi e definizioni associati ad un dato identificatore (*object_name*). Le prime due tipologie di metadati vengono utilizzate per la generazione dei grafi di dipendenza, la terza viene utilizzata per la disambiguazione sia durante la prova interattiva che durante la ricerca. La generazione di metadati viene eseguita in modalità batch e non presenta quindi interfacce HTTP o API OCaml.

Esistono inoltre altri metadati non generabili automaticamente quali quelli definiti nel modello RDF Dublin Core[11]. Sia i metadati generati automaticamente che questi ultimi sono conservati all'interno del **database di metadati** utilizzato dai servizi di livello superiore. L'interfaccia presentata dal database è quella di un DBMS in grado di rispondere a query SQL sia utilizzando client appositamente designati per il DBMS sia utilizzando API per l'interfacciamento ad esso.

Il **motore di ricerca** offre sia una interfaccia HTTP che una API OCaml che permette di eseguire diverse query espresse in un linguaggio *ad hoc* per la scrittura di query su basi di dati di conoscenza matematica formalizzata. Utilizza il database dei metadati per generare i risultati.

[11] `http://dublincore.org/`

Il **type checker** permette di verificare il corretto tipaggio (e di conseguenza la correttezza delle dimostrazioni secondo l'isomorfismo di Curry-Howard) di termini presenti nella libreria matematica o di nuovi termini che l'utente sta definendo. Offre sia una API OCaml (utilizzata ad esempio per l'implementazione di gTopLevel) che una interfaccia HTTP (utilizzata ad esempio dal sito web per permettere il type checking interattivo di documenti matematici che si stanno visualizzando).

UWOBO è la componente di HELM che permette di eseguire trasformazioni XSLT sui documenti della base documentaria. In particolare UWOBO permette l'applicazione di particolari trasformazioni, chiamate *catene*, ottenute concatenando una serie di applicazione di fogli di stile XSLT. Viene utilizzato per la resa dei documenti matematici e la navigazione all'intero del sito web, la resa dei grafi di dipendenza, ecc. Presenta una interfaccia HTTP.

HBugs non è in realtà una singola componente, ma un insieme di più componenti che permettono la generazione di *suggerimenti* durante lo svolgimento di una prova interattiva con gTopLevel. È composto internamente da un *broker* (che fornisce il punto di accesso al sistema per gTopLevel), un *client* (utilizzato da gTopLevel per interagire con il broker) e una serie di *tutor* (responsabili della generazione dei diversi tipi di suggerimenti possibili). Il broker, punto di accesso principale al sistema, presenta una interfaccia HTTP.

Il **generatore di grafi di dipendenza**, anch'esso composto in realtà da più componenti, viene utilizzato dal sito web di HELM per la rappresentazione grafica dei metadati forward e backward, ovvero delle relazioni di dipendenza tra documenti della libreria. Per la generazione dei grafi viene utilizzata una componente aggiuntiva, chiamata *uriSetQueue*, responsabile di gestire code (intese come struttura dati) di nomi logici di documenti della base documentaria. Presenta una interfaccia HTTP.

Il **sito web** `http://helm.cs.unibo.it` è il client più utilizzato per consultare il contenuto della libreria matematica di HELM. Utilizza pesantemente la quasi totalità dei servizi HELM di livello 2 per permettere all'utente di: navigare all'interno della libreria, visualizzare rappresentazioni delle dimostrazioni in diversi formati, effettuare ricerche, controllare la correttezza di termini, generare grafi di dipendenza.

gTopLevel è il proof assistant implementato nell'ambito di HELM. Permette all'utente di dimostrare interattivamente nuovi termini ed eventualmente di aggiungerli alla libreria. Offre una interfaccia grafica che è in grado di visualizzare con simbologia matematica (a differenza dell'approccio testuale utilizzato da molti altri proof assistant) le componenti della dimostrazione corrente e di sele-

zionarli come argomenti sui quali agire. Dispone di un insieme piuttosto evoluto di tattiche utilizzabili per progredire nella dimostrazione e permette all'utente di effettuare ricerche contestuali all'interno della libreria quali la ricerca dei teoremi applicabili per concludere un determinato goal. Dispone inoltre di un meccanismo di suggerimenti, basato su HBugs, che permette di sfruttare agenti che tentano di concludere uno o più goal della dimostrazione concorrentemente con il lavoro dell'utente.

In questa tesi ci occuperemo in dettaglio del getter e delle componenti di HELM di livello 2 (ad eccezione del type checker).

1.3.3 Interazioni

La maggior parte delle componenti descritte sono implementata in processi che offrono una interfaccia HTTP e che sono quindi eseguibili su host diversi raggiungibili sulla rete Internet. Riportiamo, per maggiore chiarezza, il grafo di dipendenza tra questi processi rappresentando le richieste HTTP tra loro ed i dati scambiati (fig. 1.2).

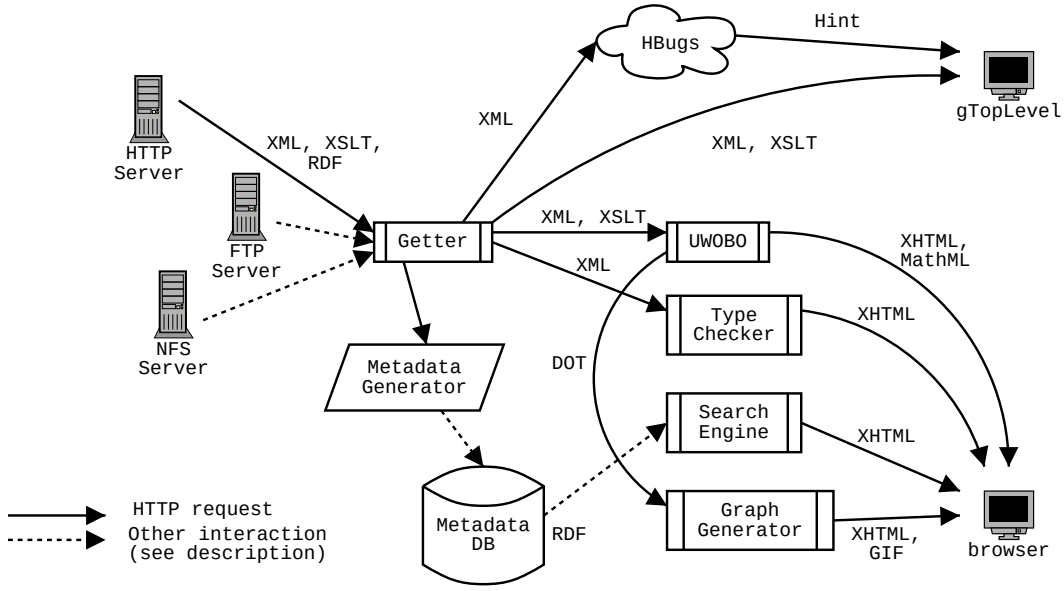


Figura 1.2: HELM: dipendenze HTTP

Il getter interagisce con i server che ospitano i documenti della base documentaria effettuando richieste HTTP, FTP o semplici accessi a file system in base alla tipologia di server. I documenti scambiati in questo modo possono es-

sere documenti XML rappresentanti conoscenza matematica formalizzata, fogli di stile XSLT descriventi trasformazioni su di essi o metadati in formato RDF.

Il generatore di metadati, eseguito in modalità batch, utilizza il getter per riempire il database dei metadati che serve, sotto forma di richieste SQL, gli stessi metadati assieme ad altri metadati inseriti manualmente al motore di ricerca.

UWOBO utilizza il getter sia per richiedere i documenti matematici che per richiedere i fogli di stile che descrivono le trasformazioni. Il type checker ed HBugs utilizzano il getter per richiedere i documenti matematici da tipare (la dipendenza tra HBugs e il type checker non è riportata nel grafo in quanto il type checker viene utilizzato mediante la sua API OCaml e non mediante richieste HTTP).

Il generatore di grafi di dipendenza utilizza UWOBO per generare una descrizione in formato Graphviz dei grafi di dipendenza e utilizza i tool associati a Graphviz per generare grafi codificati in pagine XHTML e immagini GIF da inviare al browser.

gTopLevel utilizza HBugs per ricevere suggerimenti (*Hint*) su come procedere nelle dimostrazioni ed il getter per tipare i documenti matematici utilizzati nel processo di dimostrazione.

Il browser può essere utilizzato come client per accedere a molte delle funzionalità di HELM, in particolare è possibile:

- utilizzare UWOBO per la generazione di pagine XHTML o MathML[12] corrispondenti al contenuto di parti della libreria o alla resa di documenti matematici
- utilizzare il type checker per controllare il corretto tipaggio di termini presenti all'interno della libreria
- utilizzare il motore di ricerca per cercare documenti matematici che soddisfino certi criteri
- utilizzare il generatore di grafi per analizzare le dipendenze tra documenti

1.3.4 I client

L'accesso alla libreria distribuita del progetto HELM è attualmente possibile utilizzando due HELM client:

[12] solo per browser che supportino il rendering di questo formato

- una interfaccia web (disponibile all'URL `http://helm.cs.unibo.it`)
- il proof assistant *gToplevel*

Data la stretta correlazione tra questi due client e i servizi web sviluppati, descriveremo in questa sezione la struttura di ciascuno di essi.

HELM web site

Uno degli obiettivi progettuali di HELM prevede che la libreria sia navigabile utilizzando strumenti software semplice quali normali browser grafici. Questo obiettivo è stato pienamente raggiunto realizzando il sito web di HELM[13] che funge da esempio di come realizzare punti di accesso alla libreria che pongano requisiti client side minimi.

Accedendo al sito web è infatti possibile sfruttare la quasi totalità delle funzionalità attualmente implementate del progetto HELM, in particolare:

- navigare all'interno della libreria prendendo visione degli oggetti e delle teorie che essa contiene (fig. 1.3)

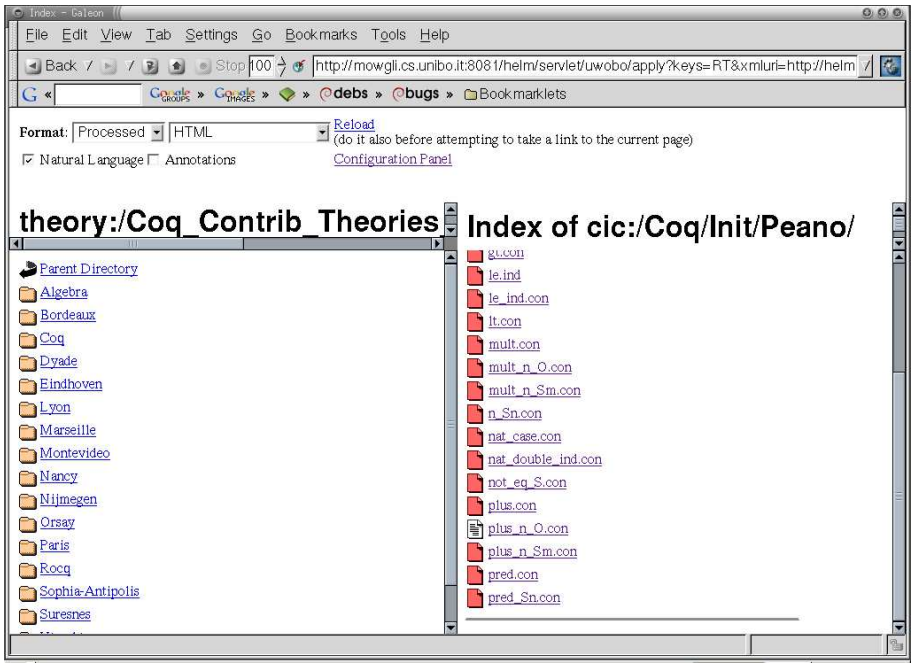


Figura 1.3: HELM web site: indice

[13] `http://helm.cs.unibo.it`

- ottenere rappresentazione matematiche dei documenti della libreria eventualmente corredate da annotazioni (fig. 1.4)

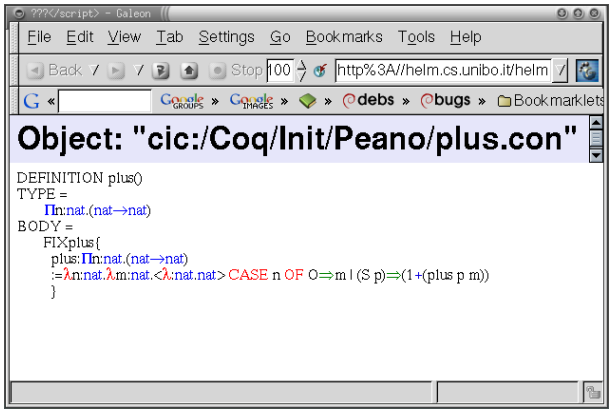


Figura 1.4: HELM web site: una definizione

- effettuare il proof checking di termini CIC verificando così la loro effettiva correttezza (fig. 1.5)

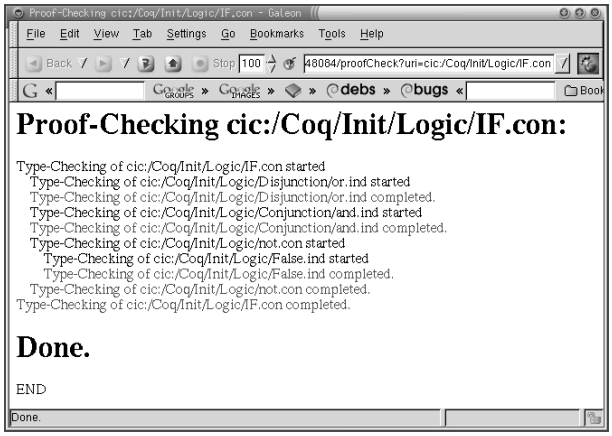


Figura 1.5: HELM web site: proof checking

- effettuare ricerche all'interno della libreria[14]
- visualizzare i metadati associati ad ogni documento (fig. 1.6)
- visionare grafi di dipendenza tra termini CIC (fig. 1.7)

[14]questa funzionalità è già stata implementata, ma non è ancora stata resa disponibile a partire dal sito web

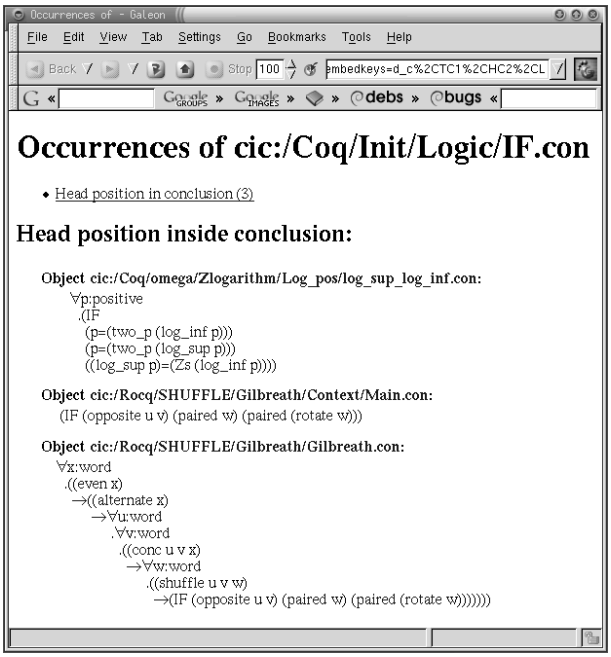


Figura 1.6: HELM web site: metadata

L'interfaccia principale di utilizzo del sito web è l'interfaccia di navigazione all'interno della libreria (fig. 1.3), le altre funzionalità sono tutte accessibili a partire da tale interfaccia.

La navigazione all'interno della libreria è possibile utilizzando due diverse modalità: la navigazione *per oggetti* e la navigazione *per teorie*. La prima richiede una conoscenza più dettagliata della disposizione degli oggetti (ovvero di definizioni, teorie, variabili, ecc.) all'interno della librerie, disposizione fortemente dipendente dai moduli di esportazione utilizzati per generare la libreria. Questa interfaccia è quindi pensata per essere utilizzata dai conoscitori dei sistemi software correlati alla libreria quali Coq data che l'organizzazione degli oggetti rispecchia spesso l'organizzazione dei moduli dei sistemi di origine.

La navigazione per teorie permette invece all'utente di navigare all'interno di raggruppamenti di documenti arbitrarie costruite *ad hoc* per scopi presentazionali o di categorizzazione. Le teorie possono contenere riferimenti ad oggetti, ma anche annotazioni, testi, figure e riferimenti ad altre teorie.

Una volta identificato un oggetto o una teoria di interesse è possibile visionarlo semplicemente attivandolo come un normale link ipertestuale, questa azione attiva una nuova finestra nella quale viene riportata la resa dell'elemento selezionato.

Le altre funzionalità sono attiviabili contestualmente alla visualizzazione dei

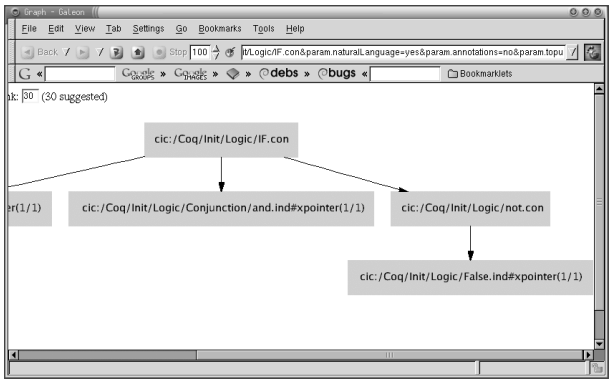


Figura 1.7: HELM web site: grafo di dipendenza

singoli oggetti selezionando link analoghi.

Architettura Le funzionalità dell'interfaccia web sono implementate basandosi sui servizi di livello 2 (sez. 1.3.1, fig. 1.1). Più in dettaglio, le funzionalità implementate nell'interfaccia web possono essere rappresentate come una pipeline HTTP all'interno della quale ogni componente svolge il ruolo di server HTTP nei confronti delle componenti di livello superiore e di client HTTP nei confronti delle componenti di livello inferiore. In fig. 1.8 è rappresentata la pipeline delle funzionalità di resa utilizzate sia per la rappresentazione grafica di oggetti e teorie (fig. 1.4) che per la visualizzazione delle due modalità di navigazione della libreria (fig. 1.3).

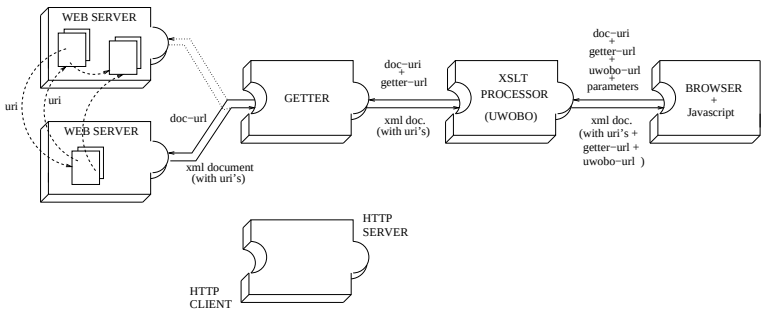


Figura 1.8: HELM web site: pipeline HTTP

L'accesso principale alla pipeline utilizzato dal browser dell'utente è l'accesso ad UWOBO. Il sito di HELM in effetti non fa altro che proporre all'utente appositi link che invocano UWOBO passando la URI del documento di base soggetto della catena di trasformazioni, il nome dei fogli di stile da applicare

e altri parametri quali il formato finale della trasformazione e i parametri da passare ai fogli di stile che lo richiedono.

Nel caso di navigazione della libreria la URI del documento base corrisponde ad una opportuna richiesta al getter che restituisce l'indice di un ramo della libreria. Nel caso di resa di un termine o di una teoria il documento di base è invece una URI che identifica il termine o la teoria all'interno della libreria di HELM.

UWOBO utilizza poi le URI ricevute per richiedere al getter i documenti a cui applicare le trasformazioni e anche gli stessi stylesheet da applicare nell'ordine richiesto. Il getter a sua volta risolve le URI in URL e scarica i documenti dai server di origine per poi restituirli ad UWOBO.

Una volta completata la catena di trasformazioni, UWOBO restituisce all'utente l'output richiesto.

L'interfaccia web contiene poi anche una parte client side, realizzata in Javascript, che implementa alcune funzionalità a "contorno" quali la selezione del formato di output e l'espansione o compressione di parti degli alberi di prova.

Utilizzando questa parte client side è inoltre possibile selezionare i componenti della pipe HTTP. È possibile ad esempio selezionare quale getter e quale UWOBO utilizzare. Scegliendo un getter diverso da quello di default è possibile consultare "viste" diverse della libreria. Scegliendo un UWOBO diverso da quello di default è invece possibile, ad esempio, utilizzare fogli di stile che supportino notazione matematica aggiuntiva.

Le altre funzionalità dell'interfaccia web sono implementate con pipeline analoghe.

Trasformazioni XSLT Le trasformazioni XSLT necessarie per la resa dei documenti della libreria di HELM sono risultate molto complesse[15] perciò è risultato necessario strutturare le trasformazioni in fasi successive applicabili in cascata ai documenti di input.

Per massimizzare la modularità è stata imposta come scelta progettuale la divisione delle trasformazioni in due macrofasi:

Fase 1 CIC (XML) → MathML Content

Fase 2 MathML Content → rappresentazione finale (HTML, XHTML, MathML Presentation)

[15] attualmente sono state scritte quasi 20.000 righe di fogli di stile XSLT

Questa scelta permette una maggiore mantenibilità degli stylesheet ed ha permesso di riutilizzare alcuni stylesheet già esistenti[16]. Un ulteriore vantaggio ottenuto grazie a questa scelta è che, sebbene la prima parte di trasformazioni possa dipendere dal sistema logico di provenienza dei documenti (è ad esempio impensabile che questa prima parte sia uguale per la resa di prove di Coq, rappresentate sotto forma di lambda termini, e per la resa di prove di NuPRL, rappresentate sotto forma di alberi di sequenti), la seconda è assolutamente indipendente da essa.

Una rappresentazione schematica delle trasformazioni XSLT necessarie per la resa degli oggetti e delle teorie è riportata in fig. 1.9.

Tutte le funzionalità aggiuntive necessarie nel processo di trasformazione sono implementate in fogli di stile che sono parte della catena di trasformazioni e che possono essere utilizzati o meno a seconda dei desideri dell'utente. È infatti auspicabile che gli utilizzatori della libreria sviluppino nuovi fogli di stile che implementino funzionalità aggiuntive, quali notazioni per costrutti matematici che non sono ancora supportati.

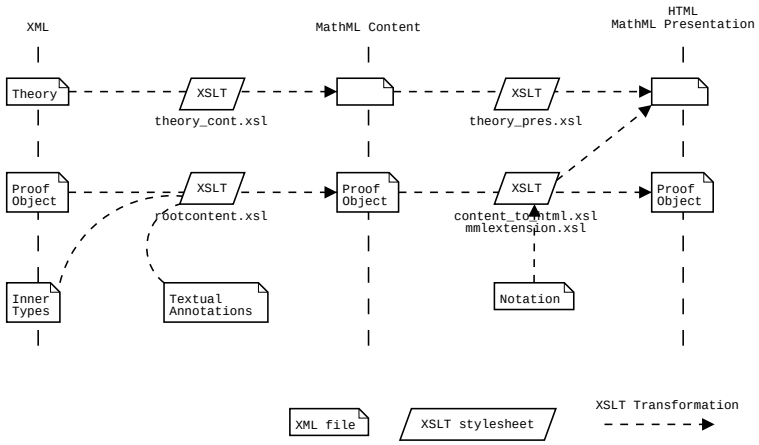


Figura 1.9: HELM web site: trasformazioni XSLT

gTopLevel

gTopLevel è il proof assistant sviluppato nell'ambito del progetto HELM.

A differenza di altri proof assistant, gTopLevel non utilizza una libreria monolitica che deve necessariamente essere installata sulla macchina sulla quale

[16]quale, ad esempio, quello di passaggio da MathML Content a MathML Presentation realizzato da Igor Rodionov del Computer Science Department della University of Western Ontario, London, Canada

viene eseguito gTopLevel, ma utilizza la libreria distribuita di HELM attraverso strumenti di accesso già visti quali il getter.

Le peculiarità di gTopLevel sono le seguenti:

- utilizzo della libreria distribuita di HELM
- resa grafica in notazione matematica delle dimostrazioni complete ed incomplete e dei goal necessari per concluderle
- discreto insieme di tattiche disponibili per la progressione nella dimostrazione corrente
- possibilità di effettuare ricerche all'interno della libreria di HELM (è ad esempio possibile cercare tutti i teoremi presenti nella libreria che possono concludere un goal)
- disambiguazione semiautomatica dei nomi di termini della libreria di HELM inseriti dagli utenti
- generazione automatica di suggerimenti per l'utente che indichino, ove possibile, come progredire nella dimostrazione corrente
- selezione interattiva mediante mouse di parti della dimostrazione e dei goal correnti con possibilità di esecuzione di azioni contestuali quali applicazione di tattiche o riapertura di parti della dimostrazione già concluse

Interfaccia grafica Uno screenshot dell'interfaccia grafica di gTopLevel è riportato in fig. 1.10.

L'interfaccia grafica è divisa in cinque componenti fondamentali:

(In)Complete proof locata nella parte alta a sinistra troviamo una rappresentazione dello stato attuale della dimostrazione. Al suo interno le metavariabili sono rappresentate utilizzando la notazione "*goal*"

Current goal locata nella parte alta a destra troviamo la finestra contenente la rappresentazione del goal corrente, al suo interno possiamo osservare la lista delle ipotesi separata dalla caonclusione del goal da una linea orizzontale. È possibile cambiare il goal corrente selezionandone un altro tra quelli disponibili. Ognuno di essi è contenuto in un tab avente etichetta "?*goal*"

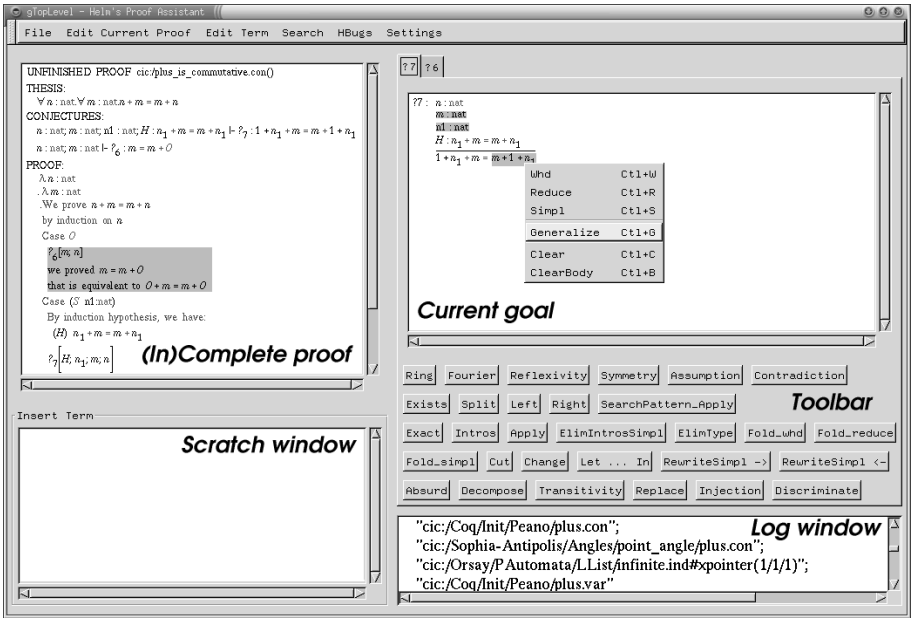


Figura 1.10: gToplevel: screenshot

Scratch window locata nella parte in basso a sinistra troviamo una area testuale[17] nella quale è possibile scrivere termini CIC ove questi siano richiesti come input di funzionalità di gTopLevel (ad esempio argomenti di tattiche, di query, ecc.)

Log Window locata nella parte bassa a destra dell'interfaccia troviamo una finestra contenente messaggi di log relativi alle azioni eseguite da gTopLevel. Attualmente questa finestra è in grado di presentare all'utente log in formato HTML

Toolbar nella parte centrale destra di gTopLevel troviamo una toolbar contenente pulsanti che attivano le varie tattiche disponibili

Sono inoltre disponibili:

- una barra di menu nella parte alta dell'interfaccia che permette di attivare funzionalità quali: caricamento e salvataggio di dimostrazioni incomplete, ricerche all'interno della libreria, personalizzazioni, attivazione e disattivazione di HBugs, ecc.

[17] è attualmente in fase di *beta testing* una estensione di gTopLevel che utilizza, al posto di questa area testuale, un editor grafico di formule matematiche basato su gtkMathView

- un sistema di menu contestuali (attivabili con il tasto destro del mouse) che permette di selezionare parti delle dimostrazioni o dei goal ed eseguire azioni su essi quali l'applicazione di tattiche che richiedono argomenti

Capitolo 2

OCamlHTTP: server HTTP in OCaml

L'insieme dei servizi web sviluppati nel corso di questa tesi è stato interamente sviluppato nel linguaggio di programmazione OCaml[1].

Tale scelta è stata dettata in parte dalla necessità di adattarsi e sfruttare componenti software già realizzate all'interno del progetto HELM scritte in tale linguaggio e in parte dai vantaggi offerti dal linguaggio stesso rispetto a linguaggi "concorrenti" (type safety, approccio funzionale, ...) sui quali non ci soffermeremo.

Come protocollo Internet sottostante ai servizi web sviluppati (ricordiamo infatti che un servizio web non necessariamente si appoggia ad uno specifico protocollo) è stato scelto HTTP, già utilizzato abbondantemente come "mezzo" di comunicazione all'interno del progetto HELMQuesta scelta è risultata vantaggiosa soprattutto per ciò che riguarda l'accessibilità dei servizi sviluppati. È infatti possibile effettuare test, sia in fase di sviluppo che a posteriori, o ancora usufruire dei servizi veri e propri avendo a disposizione un semplice browser HTTP. Per le sole finalità di debugging è comunque risultato più produttivo sviluppare una componente software ad hoc (app. A).

Le applicazioni che implementano servizi web condividono, implementativamente parlando, la parte di codice che implementa il binding al protocollo sottostante (nel nostro caso HTTP). Seguendo un ottimo principio di ingegneria del software abbiamo deciso di fattorizzare questa parte di codice in una libreria esterna il cui scopo fondamentale è quello di permettere la realizzazione di server HTTP OCaml offrendo una API "confortevole" al programmatore.

Una libreria di questo tipo, rilasciata sotto una licenza che ne permettesse la

[1] `http://www.ocaml.org`, `http://caml.inria.fr`

libera redistribuzione, il libero uso e la libera modifica non esisteva all'interno della comunità di sviluppatori del linguaggio OCaml, abbiamo quindi realizzato tale libreria che è stata denominata *OCamlHTTP*.

L'idea originale di OCamlHTTP era quella di seguire le orme di una libreria simile realizzate per il linguaggio di programmazione Perl (il modulo `HTTP::Daemon` distribuito all'interno della libreria `libwww-perl`[2], si veda [Sea02]), lo sviluppo del progetto ha poi mostrato che le possibilità offerte dall'approccio funzionale permettono di realizzare API più "confortevoli" e abbiamo spaziato anche in questa direzione.

2.1 Il protocollo HTTP

Con riferimento all'ideale stack di protocolli ISO/OSI, HTTP rientra nella categoria dei *protocolli di settimo livello*. All'interno del più concreto stack di protocolli della suite TCP/IP, HTTP rientra genericamente nell'insieme dei protocolli di livello *applicazione* al pari di altri protocolli quali FTP, SMTP, POP ecc.

Il protocollo HTTP (*HyperText Transfer Protocol*) ha come scopo e campo di applicazione la comunicazione all'interno di basi documentarie ipertestuali distribuite.

Non riporteremo una trattazione dettagliata ed esaustiva del protocollo HTTP (per la quale rimandiamo a [RFCb], [RFCa], [Ste96]), ma evidenzieremo i tratti salienti del protocollo e gli aspetti di più inerenti alla realizzazione di OCamlHTTP.

2.1.1 Generalità

HTTP è un protocollo che non prevede la presenza di uno stato ed è basato sullo scambio di messaggi testuali tra applicazioni software che partecipano alla comunicazione assumendo ruoli diversi.

I messaggi su cui si base HTTP sono quindi **testuali**. Questa scelta, comune alla quasi totalità dei protocolli di settimo livello, è una scelta di semplicità che permette di ovviare a problemi di incompatibilità di scambio di dati in formato binario che possono essere rappresentati in maniera diversa su architetture diverse o anche nelle implementazioni di sistemi operativi diversi (ad esempio ci permette di evitare problemi comuni agli sviluppatori di applica-

[2] http://www.linpro.no/lwp/

zioni TCP/IP quali l'ordinamento dei byte nella serializzazione di dati per la spedizione attraverso la rete).

I comandi propri del protocollo HTTP sono inoltre codificati secondo la codifica standard US-ASCII, permettendo così di interagire con applicazioni software HTTP anche utilizzando triviali applicazioni di manipolazione di stringhe.

Le applicazioni software che partecipano ad una comunicazione utilizzando il protocollo HTTP possono assumere diversi **ruoli** (riportiamo solamente i principali):

client una applicazione software che inizia una connessione con lo scopo di inviare richieste

server una applicazione software in grado di accettare richieste con lo scopo di elaborarle, produrre risposte e inviarle al client mittente delle richieste ricevute

origin server un particolare server sul quale risiede fisicamente una risorsa

proxy una applicazione software che svolge il doppio ruolo di client e server allo scopo di effettuare richieste per conto di uno o più client. Le richieste vengono ricevute dal proxy che provvede a inoltrarle al server e a inviare le risposte ricevute al client originale. Un proxy può modificare (non transparent proxy) o meno (transparent proxy) le richieste e le risposte che transitano attraverso di esso

gateway un particolare tipo di server che riceve richieste per conto di altri server e le soddisfa ottenendo le risorse dai server preposti. A differenza del proxy, il gateway si comporta come se fosse un origin server

tunnel una applicazione software che si comporta da veicolo per una richiesta HTTP, il tunnel non fa parte della comunicazione HTTP anche se può essere stato attivato da una richiesta HTTP

HTTP è un protocollo che non prevede la presenza di uno stato condiviso tra le applicazioni che partecipano ad una comunicazione, protocolli di questo tipo prendono il nome di **stateless**. Essendo HTTP stateless, il client è tenuto a inviare ogni volta tutte le informazioni necessarie al server affinché questi possa soddisfare una sua richiesta.

```
GET /index.html HTTP/1.1
Host: lordsoth.takhisis.org
Connection: close
```

(a) HTTP: esempio di richiesta

```
HTTP/1.1 200 OK
Date: Fri, 21 Feb 2003 14:19:23 GMT
Server: Apache/1.3.27 Debian GNU/Linux
Last-Modified: Fri, 21 Feb 2003 14:19:05 GMT
ETag: 8ae76-74-3e563559
Accept-Ranges: bytes
Content-Length: 116
Connection: close
Content-Type: text/html; charset=iso-8859-1

⟨html⟩
    ⟨head⟩
        ⟨title⟩Zack's Home page⟨/title⟩
    ⟨/head⟩
    ⟨body⟩
        ⟨h1⟩Hello, world!⟨/h1⟩
    ⟨/body⟩
⟨/html⟩
```

(b) HTTP: esempio di risposta

Figura 2.1: HTTP: esempi di messaggi

2.1.2 Formato dei messaggi

Esistono due tipi di messaggi scambiati in una comunicazione HTTP: *richieste* e *risposte*. Le richieste sono i messaggi inviati dai client ai server, le risposte sono i messaggi inviati dai server ai client. Ad ogni richiesta corrisponde una ed una sola risposta. Nelle figg. 2.1(a) e 2.1(b) sono riportati due esempi di richieste e risposte HTTP. La richiesta riportata corrisponde alla risposta riportata.

Il formato di un generico messaggio HTTP è il seguente (utilizzando una notazione simile alla notazione EBNF, per maggiori dettagli si rimanda a [RFCb])[3]:

```
message ::= start-line (message-header CRLF)* CRLF message-body?
```

La *start-line* si differenzia a seconda del tipo di messaggio: se il messaggio è una richiesta la start-line deve essere una *Request-Line*, se il messaggio è una risposta la start-line deve essere una *Status-Line*:

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Lo scopo di una Request-Line è quello di comunicare al server la tipologia di richiesta effettuata dal client, lo scopo della Status-Line è comunicare al client l'esito della sua richiesta.

[3]La stringa `CRLF` rappresenta la sequenza di caratteri US-ASCII "\r\n"

All'interno della Request-Line notiamo tre parti fondamentali: il *metodo*, una *URI* e la *versione HTTP*. Il **metodo** indica l'azione che il client richiede al server di effettuare sulla risorsa identificata dalla URI; la **URI**([RFCc]) identifica la risorsa richiesta dal client; la **versione HTTP** indica la versione del protocollo HTTP che il client sta utilizzando.

Una richiesta può includere o meno un corpo (message-body) dipendentemente dal metodo richiesto (ad esempio il metodo GET non necessità di alcun corpo, mentre i metodi PUT o POST richiedono che il client ne invii uno al server).

All'interno della Status-Line notiamo tre parti fondamentali: la *versione HTTP*, lo *status code* e la *reason phrase*. Analogamente a quanto abbiamo visto per la Request-Line, la **versione HTTP** indica la versione del protocollo HTTP in uso. Lo **status code** indica al client l'esito della richiesta, la **reason phrase** è una versione testuale della stessa informazione e comprensibile a persone piuttosto che ad applicazioni software .

Una risposta può anch'essa includere o meno un corpo dipendentemente dallo status-code.

2.1.3 Identificazione della risorsa richiesta

Ogni richiesta HTTP contiene una *request URI* il cui scopo è identificare univocamente la risorsa alla quale applicare il metodo specificato nella richiesta.

Come meccanismo di identificazione di risorse HTTP ha adottato le URI – Uniform Resource Identifier. Riportiamo qui di seguito la struttura e le componenti fondamentali di una URI. Per una descrizione dettagliata della sintassi delle stesse rimandiamo a [RFCc].

Una URI è una sequenza di caratteri in grado di identificare risorse astratte o fisiche. Le URI possono essere ulteriormente divise in *URL* (Uniform Resource Locator) e *URN* (Uniform Resource Name). Le prime identificano risorse indicando il percorso di accesso ad esse, le seconde identificano risorse indicando nomi universalmente validi e indipendenti dalle modalità di accesso alle stesse. Altra caratteristica fondamentale delle URI è la possibilità di trascrivere la stessa su normali "supporti" cartacei.

Per garantire la trascrivibilità delle URI i caratteri che le compongono sono uno stretto sottoinsieme dei caratteri US-ASCII[4]. Esistono inoltre caratteri ri-

[4]da questo sottoinsieme sono ad esempio esclusi tutti i caratteri di controllo, tipicamente non stampabili

servati utilizzati dalla specifica delle URI per separare parti significative di una URI.

Per riportare all'interno di una URI caratteri non permessi o caratteri riservati laddove non siano permessi è possibile ricorrere al meccanismo dello *URI escaping* che permette di riportare un carattere in forma *escaped* riportando in sua vece tre caratteri: il carattere "percento" ('%') seguito da due caratteri che rappresentano il codice esadecimale del carattere desiderato (esempio: le URI non ammettono sintatticamente la presenza del carattere "spazio", per rappresentare tale carattere è possibile utilizzare la sequenza %20).

La struttura sintattica più generale possibile di una URI è

⟨scheme‘‘:’’⟩:⟨schema-specific-part⟩

Lo *scheme* identifica il "tipo" di URI, l'interpretazione della *schema-specific-part* dipende interamente da esso.

Ciò nonostante molti schemi di URI condividono una sintassi generica:

⟨scheme⟩‘‘://’’⟨authority⟩⟨path⟩?⟨query⟩

La parte *authority* è tipicamente gerarchica e dipendente da entità autorizzate a gestire tale gerarchia (ad esempio le naming authority per quanto riguarda i nomi di dominio di Internet). La parte *path* rappresenta un percorso di accesso alla risorsa all'interno della *authority*. La parte *query* è una stringa di informazioni interpretabili dalla risorsa identifica dal path.

Una sintassi molto diffusa per la parte query è la seguente:

```
query ::= ‘‘?’’  binding (‘‘&’’ binding)*
      binding ::= name ‘‘=’’ value
```

nella quale ogni elemento *binding* rappresenta l'associazione di un valore ad un nome. L'insieme dei binding costituisce un *ambiente* che viene passato alla risorsa (questa convenzione viene ad esempio utilizzata dalle applicazioni che utilizzano l'interfaccia CGI – Common Gateway Interface[5]).

2.1.4 Metodi

I diversi metodi HTTP indicano al server quale azione è stata richiesta dal client sulla risorsa indicata dalla URI presente nella Request-Line.

[5] `http://hoohoo.ncsa.uiuc.edu/cgi/interface.html`

HTTP definisce alcune caratteristiche dei metodi, in particolare definisce i concetti di metodo *sicuro* e metodo *idempotente*.

Un metodo si definisce **sicuro** quando il suo effetto principale è quello di fornire accesso ad una risorsa. La distinzione tra metodo sicuri e non permette ad applicazioni quali browser o altri client di avvisare l'utente nel caso in cui si stiano utilizzando metodi non sicuri che potenzialmente possono avere altri effetti quali creare risorse o modificare risorse presenti sul server (ad esempio il metodo PUT). I metodi definiti come sicuri da HTTP 1.1 sono GET e HEAD. È importante notare che un metodo sicuro non è garantito essere scevro da effetti collaterali.

Un metodo si definisce **idempotente** quando gli effetti collaterali dell'invio di $n > 0$ richieste identiche che utilizzano tale metodo sono indistinguibili dagli effetti collaterali generati dall'invio di una singola richiesta. HTTP 1.1 definisce come idempotenti i metodi: GET, HEAD, PUT, DELETE, OPTIONS e TRACE.

HTTP 1.1 definisce otto metodi predefiniti e lascia la possibilità agli implementatori di server HTTP di definire metodi aggiuntivi non standard. I metodi predefiniti in HTTP 1.1 sono i seguenti:

OPTIONS indica al server che il client richiede informazioni sulle opzioni di comunicazioni disponibili lungo il percorso richiesta/risposta per la risorsa indicata nella request URI

GET richiede al server l'invio di una risorsa indicata dalla request URI. La request URI può identificate una risorsa fisica presente sul server o può contenere parametri da passare ad una applicazione in esecuzione sul server per generare la risorsa richiesta

HEAD è analogo al metodo GET con la differenza che il server non invia la risorsa stessa al client, ma solamente gli header ad essa associati (ciò implica che la risposta non conterrà alcun corpo)

POST indica al server di accettare la ricezione dei dati inviati dal client nel corpo della richiesta come dati per la risorsa indicata nella request URI, tali dati possono ad esempio essere dati da elaborare per una applicazione remota, annotazioni riguardanti una risorsa, dati da aggiungere ad una base di dati ecc.

PUT indica al server di creare la risorsa identificata dalla Request URI e di utilizzare il corpo della richiesta come suo contenuto

DELETE richiede al server la cancellazione della risorsa identificata dalla request URI

TRACE questo metodo è utilizzato principalmente per debugging e richiede al server di inviare come risposta al client il corpo della richiesta appena ricevuta

CONNECT metodo riservato per proxy che possono dinamicamente assumere il ruolo di tunnel

2.1.5 Codici di stato

HTTP 1.1 definisce 40 diversi codici di ritorno (*status code*) predefiniti e relativa semantica, utilizzabili dal server per comunicare l'esito della elaborazione della richiesta del client, e permette inoltre agli implementatori di server web di definirne di nuovi.

Gli status code definiti da HTTP 1.1 sono divisi in cinque famiglie (differenziate dalla prima cifra dello status code) che descriviamo brevemente. Per una lista completa degli status code si veda [RFCb] oppure la API di OCamlHTTP.

Informational utilizzati per inviare al client messaggi temporanei durante lo svolgimento della elaborazione della richiesta da parte del server

Successfull indicano al client che il server ha ricevuto, compreso e accettato la sua richiesta

Redirection indicano che sono necessarie altre azioni da parte del client per portare a termine la richiesta

Client error la richiesta non può essere soddisfatta a causa di un errore commesso dal client, tipicamente danno indicazioni al client sull'origine dell'errore in modo che possa essere effettuata una richiesta corretta successivamente

Server error la richiesta non può essere soddisfatta a causa di un errore verificatosi presso il server

2.1.6 Header

Ogni messaggio HTTP può contenere una lista di header atti a trasportare meta informazioni. Le informazioni trasportate come valori degli header possono riguardare:

- la trasmissione del messaggio
- l'entità contenuta nel messaggio (corpo)
- la richiesta effettuata
- la risposta effettuata

A seconda delle caratteristiche del messaggio possono essere presenti o meno header di diversa natura, in particolare: gli header inerenti la trasmissione del messaggio possono sempre comparire, gli header inerenti all'entità trasmessa possono comparire solamente se il messaggio trasmesso contiene un corpo, gli header inerenti la richiesta e la risposta possono comparire solamente sulle rispettive tipologie di messaggi.

La struttura sintattica di un header è composta da una coppia di valori *nome*, *valore* (eventualmente assente) separati dal carattere ":".

```
message-header = field-name ‘‘:’’  [ field-value ]
```

2.1.7 Connessioni

Come abbiamo visto, il protocollo HTTP è basato sul modello richiesta/risposta: un client effettua una o più richieste ad un server che risponde inviando al client una o più risposte, ad ogni richiesta di un client corrisponde una ed una sola risposta del server.

Una "sessione", composta dall'invio di richieste dal client al server e dalla ricezione delle corrispondenti risposte da parte del client, prende il nome di *connessione* HTTP. Ad ogni connessione HTTP corrisponde di fatto una singola connessione TCP iniziata attivamente dal client verso il server e chiusa o dal client o dal server al termine della comunicazione.

L'overhead di comunicazione richiesto per stabilire una connessione TCP e per chiuderla a posteriori ha fatto si che il concetto di connessione sia mutato nelle differenti versioni del protocollo HTTP. Le differenze fondamentali sono state comunque apprezzabili nel passaggio da HTTP 1.0 a HTTP 1.1.

In HTTP 1.0 infatti ogni connessione era composta dall'invio di una singola richiesta dal client al server e dall'invio di una singola risposta dal server al client. Considerando che tipicamente una singola pagina Web, per essere visualizzata correttamente, richiede l'esecuzione di molti cicli richiesta/risposta (ad esempio è necessario effettuare una richiesta per ogni frame o per ogni immagine presente all'interno di una pagina HTML), l'overhead di apertura e chiusura delle connessioni era considerevole.

Per ovviare a questo overhead in HTTP 1.1 il concetto di connessione è stato ampliando permettendo a client e server di accordarsi per non chiudere la connessione dopo la ricezione della prima risposta o ancora per permettere al client di inviare una serie di richieste sulla stessa connessione prima ancora di avere ricevuto una singola risposta.

Il primo approccio prende il nome di *connessione persistente* (fig. 2.2(b)), il secondo di *pipelining* (fig. 2.2(d)).

Affinché sia possibile mantenere le connessioni persistenti è necessario che ogni messaggio HTTP scambiato tra client e server sia di una dimensione nota.

2.2 Architettura

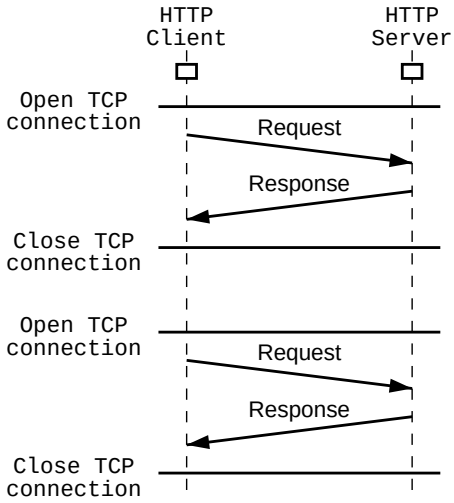
La vita di un server HTTP può essere divisa nelle fasi di: avvio, attesa di richiesta, gestione della richiesta, invio della risposta ed eventualmente terminazione (fig. 2.3).

Nella fase di **avvio** il server HTTP esegue varie operazioni di inizializzazione che necessariamente includono il *binding* di un indirizzo IP e di una porta ben definita in modo che il server possa essere contattato dai client. Altre operazioni di inizializzazione "interessanti" includono la configurazione della directory nota come *document root* (directory che viene considerata come la radice del filesystem gestito dal server, i documenti presenti in tale directory vengono considerati relativi alla radice di documenti serviti dal server) e dei parametri di accesso per quanto riguarda l'autenticazione richiesta ai client che accederanno al server.

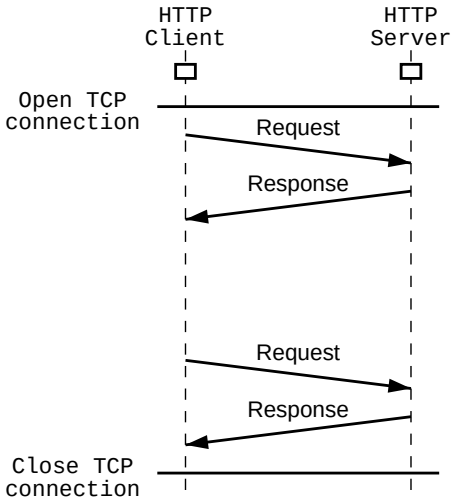
Il server poi resta in attesa della ricezione di una richiesta HTTP, una volta ricevuta una richiesta questa viene processata dal server controllando che sia ben formata e che rispetti quindi lo standard HTTP e viene generate una risposta ad essa corrispondente. Tale risposta viene inviata al client e il server si mette nuovamente in attesa di una richiesta HTTP.

La API di OCamlHTTP permette al programmatore di gestire tutte le fasi della vita di un server HTTP, in particolare:

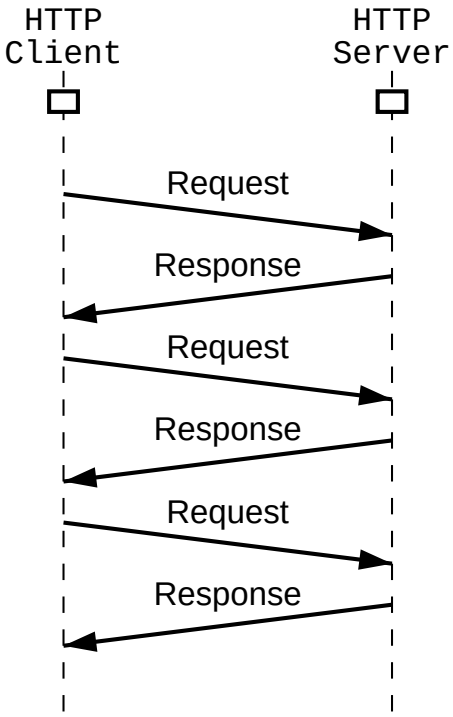
- permette di stabilire i parametri di avvio del server quali l'indirizzo IP e la porta su cui "ascoltare", la document root, un eventuale timeout dopo il quale chiudere le connessioni dei client
- permette di analizzare il contenuto delle richieste dei client
- permette di generare risposte HTTP e di inviarle ai client



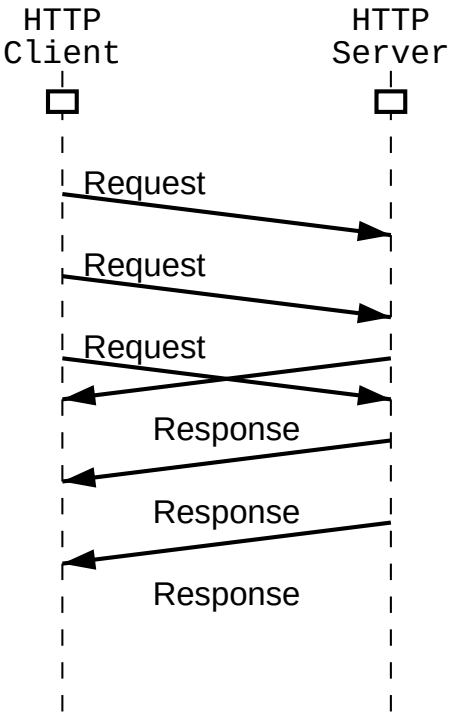
(a) HTTP: connessioni *non* persistenti



(b) HTTP: connessioni persistenti



(c) HTTP senza pipelining



(d) HTTP con pipelining

Figura 2.2: Tipologie di connessioni HTTP

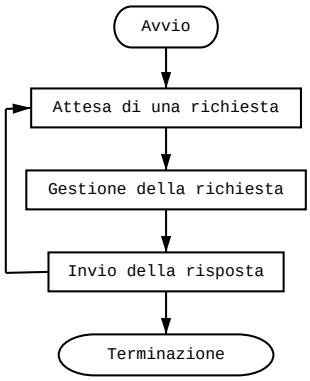


Figura 2.3: Vita di un server HTTP

- permette di segnalare al server che deve terminare il suo normale flusso di esecuzione

Le funzionalità della API possono essere distinte in alcuni gruppi funzionali:

- inizializzazione di demoni HTTP
- formalizzazione dei messaggi HTTP
- funzioni di utilità per l'invio di risposte HTTP

2.2.1 Inizializzazione di demoni HTTP

È possibile inizializzare demoni HTTP utilizzando OCamlHTTP secondo due diversi approcci:

- l'approccio *funzionale*
- l'approccio *imperativo*

In entrambi gli approcci è possibile specificare i parametri di inizializzazione del demone HTTP (per ognuno di essi è previsto un *default* nel caso in cui essi non siano specificati), i parametri di inizializzazione attualmente supportati sono:

- indirizzo IP sul quale il demone HTTP si porrà in ascolto
- porta TCP sulla quale il demone HTTP si porrà in ascolto

- tempo di vita per la gestione di ogni richiesta HTTP, se una richiesta HTTP non è ancora stata servita allo scadere del timeout per un qualsiasi motivo (la richiesta non è stata inviata interamente dal client, il server la sta ancora processando, ecc.) la connessione con il client viene chiusa dal server
- modalità di gestione delle richieste (si veda sez. 2.2.2)
- document root

Approccio funzionale

Utilizzando questo approccio il *main loop* della vita di un demone HTTP (si veda fig. 2.3) è gestito internamente da OCamlHTTP. Al programmatore è richiesto di fornire una funzione di *callback* che viene invocata ogni qual volta una richiesta HTTP viene ricevuta dal demone HTTP.

È importante osservare che le callback vengono invocate solamente dopo che le richieste HTTP sono state processate da OCamlHTTP e valutate corrette secondo lo standard HTTP. Nel caso in cui vengano ricevute richieste non ben formate OCamlHTTP provvede autonomamente a inviare risposte di errore appropriate[6] al client. In questo modo l'implementazione delle callback può evitare di effettuare test di correttezza sulle richieste ricevute e lavorare su strutture più ad alto livello che non le semplici stringhe di testo componenti la richiesta.

I parametri ricevuti in input dalla callback sono diversi a seconda della funzione che si utilizza per l'inizializzazione del server, in generale è possibile definire:

- callback *semplici* (Http_daemon.start)
- callback *avanzate* (Http_daemon.start')

Le prime sono pensate per la scrittura di demoni che non necessitino di accedere a informazioni dettagliate sulle richieste ricevute. Le seconde sono invece più potenti e permettono di accedere a tutti i dettagli delle richieste ricevute.

In particolare le **callback semplici** hanno il seguente prototipo:

```
string -> (string * string) list -> out_channel -> unit
```

[6]nella maggior parte dei casi vengono spedite risposte aventi codice di ritorno `400 - Malformed request`, ma possono anche essere generati codici di ritorno quali `505 - HTTP Version not supported`, `501 - HTTP Method not implemented` o altre

Notiamo innanzitutto che le callback non ritornano alcun valore interessante per OCamlHTTP, ritornano infatti il tipo `unit`.

Il primo argomento delle callback semplici è la componente *path* della request URI (è da notare come questa URI sia già stata controllata da OCamlHTTP e che si possa pertanto considerare corretta per quanto riguarda la sola sintassi delle URI).

Il secondo argomento è una decodifica dell'ambiente passato dal client (si veda sez. 2.1.3). La callback riceve quindi in questo argomento una lista di coppie di stringhe: il primo elemento di ogni coppia rappresenta il nome di un elemento dell'ambiente passato dal client, il secondo elemento di ogni coppia rappresenta il valore ad esso associato.

Attualmente OCamlHTTP supporta i metodi GET e POST di HTTP 1.1. Nel caso del metodo GET la lista di coppie contiene solamente la rappresentazione dell'ambiente decodificato dalla query string contenuta nella request URI. Nel caso del metodo POST la lista contiene l'ambiente decodificato dal corpo della richiesta *concatenato* all'ambiente decodificato dalla query string.

Il terzo e ultimo argomento è un output channel (tipo definito nel modulo `Pervasives` della libreria standard OCaml) collegato al client remoto. È possibile scrivere direttamente su questo canale per inviare la risposta al client oppure utilizzare le funzioni offerte da OCamlHTTP per un approccio più ad alto livello di astrazione.

La scelta di rendere visibile il canale connesso al client è stata dettata da motivi di *scalabilità*. All'interno del progetto HELM infatti è risultato necessario trattare risposte HTTP che occupano una quantità considerevole di memoria. Lasciando il canale visibile nelle callback è possibile scrivere callback che inviano incrementalmente la risposta al client senza necessità di mantenere l'intera risposta in memoria. Sottolineiamo comunque che è possibile *sia* scrivere direttamente sul canale *che* utilizzare le funzioni di astrazione offerte da OCamlHTTP.

A titolo di esempio riportiamo la callback semplice di un demone HTTP che si limita a stampare su *standard error* il valore del parametro "x" ricevuto (senza rispondere al client, dal momento che non sono ancora state mostrate le funzioni per generare risposte HTTP):

```
let callback _ pars _ = prerr_endline (List.assoc x pars)
```

Le **callback avanzate** hanno invece il seguente prototipo:

```
Http_types.request -> out_channel -> unit
```

Il valore di ritorno e l'ultimo parametro ricevuto dalle callback avanzate sono analoghi a quanto visto per le callback semplici e non richiedono quindi ulteriori discussione.

Per quanto riguarda il primo parametro notiamo invece che non è più un tipo primitivo di OCaml, ma un oggetto la cui classe è definita in OCamlHTTP. In particolare si tratta della formalizzazione delle richieste HTTP implementate nel modulo `Http_request`. Questo oggetto permette di ottenere molte più informazioni a riguardo di una richiesta HTTP di quanto sia possibile con le callback semplici. In particolare è possibile accedere anche distintamente all'ambiente ricevuto all'interno della query string e a quello passato nel corpo della richiesta (nel caso di metodo POST) e sono inoltre disponibili molte informazioni aggiuntive quali gli header della richiesta, il corpo "grezzo" della richiesta, gli indirizzi IP di client e server coinvolti nella comunicazione, il metodo della richiesta e il path della risorsa richiesta.

A titolo di esempio riportiamo la callback avanzata equivalente a quella dell'esempio visto per le callback semplici:

```
let callback req _ = prerr_endline (req#param x)
```

Terminazione è possibile per una callback richiedere l'interruzione del ciclo infinito di vita di un demone HTTP ed il passaggio allo stato di *terminazione* (fig. 2.3).

Questa informazione viene passata dalle callback a OCamlHTTP sollevando l'eccezione *Quit*, definita nel modulo Http_types:

```
exception Quit
```

OCamlHTTP sorveglia l'esecuzione delle callback con il costrutto OCaml *try .. with* verificando se eccezioni Quit vengano sollevate o meno. Nel caso in cui ciò accada, attende la terminazione di tutte le callback attualmente in esecuzione e termina l'esecuzione del ciclo di vita del demone HTTP (ciò implica che le funzioni utilizzate per inizializzare il demone HTTP, quali Http_daemon.start, ritornano).

Le eccezione Quit sollevate dalle callback vengono gestite interamente da OCamlHTTP e non vengono ulteriormente propagate al di fuori delle invocazioni delle funzioni di inizializzazione.

Approccio imperativo

L'approccio imperativo di OCamlHTTP è l'approccio più vicino a quello del modulo `HTTP::Daemon` della libreria Perl `libwww-perl`.

A differenza del caso funzionale, in questo approccio il main loop caratterizzante la vita di un server web è gestito dal programmatore che deve preoccuparsi di inizializzare il server e di gestire l'arrivo di una richiesta invocando appositi metodi.

Nella pratica un tipico demone HTTP sviluppato utilizzando l'approccio imperativo di OCamlHTTP è sviluppato secondo lo schema descritto nel seguente listato scritto in pseudocodice OCaml:

```
1  (* daemon's initialization *)
2  let d = new Http_daemon.daemon (* parameters *) () in
3  while true do
4    let (req, conn) =
5      d#getRequest (* wait for valid request *)
6    in
7    let res = (* create a new response *) in
8    conn#respond_with res;
9    conn#close
10 done
```

Viene innanzitutto creato un oggetto di tipo `Http_daemon.daemon` (linea 2) il prototipo per l'istanziazione di oggetti di questo tipo permette di specificare l'indirizzo IP e la porta TCP sulla quale il demone si porrà in ascolto:

```
class daemon:
   ?addr:  string -> ?port:  int -> unit ->
      Http_types.daemon
```

L'interfaccia della classe è molto semplice e presenta solamente due metodi: `accept` e `getRequest`:

```
1  class type daemon =
2    object
3      method accept: Http_types.connection
4      method getRequest: Http_types.request * Http_types.connection
5    end
```

Il primo metodo (accept) è bloccante e ritorna un valore solamente dopo che il demone ha ricevuto una connessione da un client HTTP. L'oggetto ritornato dal metodo accept ha tipo `Http_types.connection` e permette al programmatore di ottenere le richieste[7] inviate dai client:

[7] il plurale fa riferimento ad una futura estensione della libreria che supporti richieste HTTP pipelined, sarà quindi possibile utilizzare il metodo più volte sullo stesso oggetto connection per accedere a tutte le richieste contenute nella pipeline

```
class type connection =
  object
    method getRequest: Http_types.request option
    method respond_with: Http_types.response -> unit
    method close: unit
  end
```

Il metodo `getRequest` di questo oggetto restituisce una richiesta (si veda sez. 2.2.3) nel caso in cui il client abbia inviato una richiesta ben formata oppure `None` in caso contrario. Il metodo `respond_with` dell'oggetto connection permette di inviare un oggetto risposta come risposta HTTP al client. È inoltre presente il metodo `close` nel caso in cui si voglia terminare la connessione con il client senza inviare risposte. È importante sottolineare che, una volta invocato questo metodo, non è più possibile comunicare con il client utilizzando l'oggetto connection sul quale è stato invocato il metodo.

Il secondo metodo dell'oggetto daemon (getRequest) è un metodo bloccante che, a differenza di quanto visto per accept, ritorna solamente dopo che una richiesta ben formata è stata ricevuta dal server. Utilizzando questo metodo quindi si delega all'oggetto daemon la gestione delle richieste mal formate perdendo però la possibilità di distinguere tra il momento della ricezione di una connessione e il momento della ricezione di una richiesta. Il metodo getRequest ritorna una coppia: il primo elemento è la richiesta ricevuta dal client, il secondo elemento è un oggetto di tipo connessione, analogo a quelli già visti, utilizzabile per inviare risposta al client.

2.2.2 Modalità di gestione dei client

Un server HTTP può dovere sostenere un carico di lavoro di molte richieste per secondo, risulta quindi spesso necessario implementare server HTTP che siano in grado di gestire più richieste contemporaneamente.

OCamlHTTP permette di gestire più richieste concorrentemente e anche la comunicazione tra le callback che stanno gestendo le richieste. Queste possibilità sono lasciate interamente al programmatore nell'approccio imperativo che deve preoccuparsi di creare nuovi thread o nuovi processi quando lo ritiene opportuno dal momento che il main loop del server HTTP non è gestito da OCamlHTTP.

Nell'approccio funzionale invece, OCamlHTTP introduce il concetto di *modalità* di gestione dei client. Tale modalità viene specificata mediante il parametro `mode` delle funzioni di inizializzazione dei demoni HTTP (ad esempio `Http_daemon.start`). Il parametro mode è opzionale in tutte le funzioni di inizializzazione, il suo valore di default è `‘Fork`.

Esistono tre diverse modalità di gestione dei client:

- `'Single`
- `'Fork`
- `'Thread`

Nella modalità **'Single** le richieste HTTP sono servite sequenzialmente secondo il classico modello *busy-wait*: non più di una richiesta contemporanea viene gestita. Nel caso in cui arrivi una richiesta prima che sia terminata la gestione della precedente, il client viene messo in attesa, la coda di attesa del server HTTP è quella specificata dal parametro *backlog* della chiamata di sistema POSIX `listen` invocata all'inizializzazione del server (attualmente OCamlHTTP imposta tale valore a 10, di conseguenza la coda di attesa può contenere al più 10 client).

In questa modalità si può quindi assumere che il codice che implementa la callback del server HTTP sia eseguito sempre dallo stesso processo e mai in contemporanea ad altre istanze dello stesso codice.

Nella modalità **'Fork** viene creato un nuovo *processo* per ogni richiesta HTTP ricevuta, a tale processo viene delegata l'esecuzione della callback alla quale viene passato come parametro il canale di output collegato al client.

I processi sono creati invocando la chiamata di sistema POSIX `fork`. Ognuno di essi viene quindi eseguito in una copia dello spazio di memoria del processo che ha inizializzato il server. Non vi è alcun passaggio di informazioni implicito tra i processi che eseguono le callback e nemmeno tra i processi che eseguono le callback ed il processo che esegue il main loop del server HTTP.

È possibile mettere in comunicazione tali processi solamente utilizzando le usuali primitive Unix di comunicazione interprocesso (socket, memoria condivisa, semafori, ecc.).

La modalità **'Thread** si comporta in linea di principio analogamente alla modalità 'Fork con la differenza che per la gestione di ogni richiesta HTTP viene creato (utilizzando la chiamata di sistema POSIX `clone`[pth96]) un nuovo *thread* invece di un nuovo processo.

Ogni thread viene creato utilizzando la API di programmazione di thread offerta da OCaml e viene eseguito nello stesso spazio di memoria del processo che gestisce il main loop del server HTTP. Risulta quindi necessario proteggere le strutture globali con sezioni critiche per evitare inconsistenze di dati.

I thread incaricati di gestire le richieste HTTP e il thread incaricato di gestire il main loop possono comunicare tra loro utilizzando le primitive di

comunicazione interthread offerte dalla libreria standard OCaml quali mutue esclusioni (modulo `Mutex`), condizioni (modulo `Condition`) e eventi (modulo `Event`).

2.2.3 Formalizzazione dei messaggi HTTP

La API di OCamlHTTP distingue tre diversi tipi di messaggi HTTP:

1. messaggi HTTP
2. richieste HTTP
3. risposte HTTP

Ad ogni tipo di messaggi è abbinata una classe OCaml. Le relazioni di ereditarietà tra le classi così ottenute sono descritte in fig. 2.4.

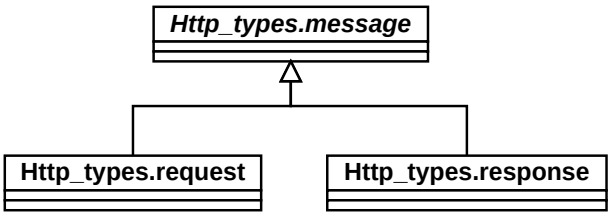


Figura 2.4: Messaggi HTTP - Ereditarietà

La formalizzazione dei messaggi di OCamlHTTP rispecchia la definizione di messaggi HTTP riportata in [RFCb]. Richieste e risposte HTTP sono infatti rappresentate come classi derivate dalla classe virtuale "messaggio HTTP".

Messaggi HTTP

La classe abbinata ad un generico messaggio HTTP in OCamlHTTP è la classe *virtuale* `Http_message.message`. Questa classe non è istanziabile, ma è utilizzata per la definizione delle classi richiesta e risposta.

L'interfaccia di questa classe include metodi che possono essere raggruppati funzionalmente come segue:

- accesso alla versione HTTP
- accesso al corpo del messaggio
- accesso agli header

- accesso agli indirizzi di client e server
- serializzazione del messaggio

```
1  class type message = object
2
3        (* HTTP version access *)
4      method version: version option
5      method setVersion: version -> unit
6
7        (* message body access *)
8      method body: string
9      method setBody: string -> unit
10     method bodyBuf: Buffer.t
11     method setBodyBuf: Buffer.t -> unit
12     method addBody: string -> unit
13     method addBodyBuf: Buffer.t -> unit
14
15       (* headers access *)
16     method addHeader: name:string -> value:string -> unit
17     method addHeaders: (string * string) list -> unit
18     method replaceHeader: name:string -> value:string -> unit
19     method replaceHeaders: (string * string) list -> unit
20     method removeHeader: name:string -> unit
21     method hasHeader: name:string -> bool
22     method header: name:string -> string
23     method headers: (string * string) list
24
25       (* client address access *)
26     method clientSockaddr: Unix.sockaddr
27     method clientAddr: string
28     method clientPort: int
29
30       (* server address access *)
31     method serverSockaddr: Unix.sockaddr
32     method serverAddr: string
33     method serverPort: int
34
35       (* message serialization *)
36     method toString: string
37     method serialize: out_channel -> unit
38
39   end
```

I metodi di **accesso alla versione HTTP** (`version`, `setVersion`, righe 4-5) permettono di ottenere e modificare la versione HTTP associata al messaggio.

I metodi di **accesso al corpo del messaggio** (`body`, `setBody`, `bodyBuf`, `setBodyBuff`, `addBody`, `addBodyBuf`, righe 8-13) permettono di ottenere il contenuto del messaggio HTTP e di reimpostarlo. È possibile accedere al corpo del messaggio sia in formato stringa sia in formato buffer (`Buffer.t`) è inoltre possibile reimpostare interamente il contenuto del messaggio o di comporlo incrementalmente.

I metodi di **accesso agli header** (`addHeader`, `addHeaders`, `replaceHeader`, `replaceHeaders`, `removeHeader`, `hasHeader`, `header`, `headers`, righe 16-23) permettono un controllo completo sugli header del messaggio HTTP. È possibile accedere ad header già impostati, aggiungerne dei nuovi, rimuoverne ed effettuare controlli sull'esistenza di alcuni di essi.

I metodi di **accesso agli indirizzi di client e server** (`clientSockaddr`, `clientAddr`, `clientPort` e i corrispettivi per il server, righe 25-33) permettono di accedere all'indirizzo IP di client e server sia in formato stringa che in formato `Unix.sockaddr` e di accedere alla porta TCP di client e server. L'accesso a tali valori è in sola lettura. È importante ricordare che per quanto riguarda le richieste il client corrisponde al mittente del messagio ed il server corrisponde al destinatario, per quanto riguarda le risposte la situazione risulta invertita.

I metodi inerenti alla **serializzazione del messaggio** sono due: `toString` e `serialize` (righe 36-37). Il primo restituisce una rappresentazione testuale del messaggio, il secondo scrive direttamente tale rappresentazione su un canale di output.

Richieste HTTP

La classe di OCamlHTTP abbinata ad una generica richiesta HTTP è `Http_request.request`. Questa classe, a differenza di quella abbinata ai messaggi HTTP, è concreta e può essere istanziata fornendo come unico argomento un canale di input dal quale viene letta e processata una richiesta HTTP ben formata:

```
class request:
   in_channel ->
      Http_types.request
```

I metodi dell'ogetto così ottenuto possono essere divisi funzionalmente in gruppi:

- metodi ereditati da message
- accesso al metodo

- accesso alla request URI
- accesso all'ambiente

```
1  class type request = object
2
3      inherit message
4
5      method meth: meth
6
7      method uri: string
8      method path: string
9
10     method param: ?meth:meth -> string -> string
11     method paramAll: ?meth:meth -> string -> string list
12     method params: (string * string) list
13     method params_GET: (string * string) list
14     method params_POST: (string * string) list
15
16   end
```

request è una classe derivata da message (riga 3), in quanto tale sono disponibili sulle sue istanze tutti i metodi visti per la classe message.

Il metodo `meth` (riga 5) permette di accedere al metodo HTTP abbinato ad ogni richiesta.

I metodi di **accesso alla request URI** sono due: `uri` e `path` (righe 7-8). Il primo restituisce l'intera request URI mentre il secondo restituisce solamente la componente path in essa contenuta.

I metodi di **accesso all'ambiente** (`param`, `paramAll`, `params`, `params_GET`, `params_POST`) permettono di accedere all'ambiente costruito a partire dalla query string (per il metodo GET) e dal corpo del messaggio (per il metodo POST). I metodi più utilizzati sono `param` e `paramAll` che permettono di ottenere il valore di un parametro dell'ambiente o la lista di tutti i suoi valori nello stesso cercandolo o nella concatenazione degli ambienti ottenuti dalla query string e dal corpo del messagio o in uno solo di essi scelto passando il parametro `meth`. I metodi rimanenti permettono di accedere agli ambienti interamente sotto forma di lista di coppie nome, valore. Tutti gli accessi a dati contenuti in una richiesta HTTP avvengono in modalità di sola lettura.

Risposte HTTP

Le risposte HTTP sono formalizzate in OCamlHTTP nella classe `Http_response.response`. Per questa classe valgono tutte le considerazioni

già riportare per request. È possibile istanziarla senza passare alcun parametro significativo al costruttore o impostare direttamente molti dei suoi attributi quali corpo, header, versione del protocollo, ecc.:

```
class response:
   ?body:string ->
   ?headers:(string * string) list -> ?version:  version ->
   ?clisockaddr:  Unix.sockaddr -> ?srvsockaddr:  Unix.sockaddr ->
   ?code:int -> ?status:Http_types.status -> unit ->
      Http_types.response
```

All'interno dei metodi offerti dalla classe possiamo distinguere due gruppi funzionali:

- accesso alla status line
- accesso agli header

```
1  class type response = object
2
3      inherit message
4
5      method code: int
6      method setCode: int -> unit
7      method status: status
8      method setStatus: status -> unit
9      method reason: string
10     method setReason: string -> unit
11     method statusLine: string
12     method setStatusLine: string -> unit
13     method isInformational: bool
14     method isSuccess: bool
15     method isRedirection: bool
16     method isClientError: bool
17     method isServerError: bool
18     method isError: bool
19
20     method addBasicHeaders: unit
21     method contentType: string
22     method setContentType: string -> unit
23     method contentEncoding: string
24     method setContentEncoding: string -> unit
25     method date: string
26     method setDate: string -> unit
27     method expires: string
28     method setExpires: string -> unit
```

```
29      method server: string
30      method setServer: string -> unit
31
32    end
```

I metodi per l'**accesso alla status line** (righe 5-18) permettono di accedere in due diversi modi al codice di ritorno della richiesta HTTP (sia utilizzando il codice numerico che utilizzando valori più ad alto livello definiti in `Http_types`), alla reason phrase e alla status line nella sua interezza. Sono inoltre presenti sei predicati (righe 13-18) che permettono di valutare quale sia la classe di appartenenza del codice di ritorno della richiesta come definite in [RFCb].

I metodi di **accesso agli header** (righe 20-30) permettono, addizionalmente a quanto già permesso dai metodi di accesso agli header ereditati da message, di aggiungere alcuni header tipici di ogni risposta (riga 20)[8]. Sono inoltre presenti un insieme di metodi di comodità che permettono di accedere in lettura e scrittura a header di uso comune per le risposte quali: `Content-Type`, `Content-Encoding`, `Date`, `Expires` e `Server`.

2.3 Invio di risposte ai client

Le modalità con le quali è possibili rispondere ai client in OCamlHTTP sono differenti a seconda dell'approccio che si è scelto.

Nell'approccio imperativo (sez. 2.2.1) il programmatore non ha visibilità del canale di output connesso al client. È possibile rispondere solamente utilizzando il metodo respond_with dell'oggetto connection che riceve come unico argomento una istanza della classe Http_request.request.

Una tipica risposta in questo approccio prevede quindi che il programmatore crei un oggetto request e che lo invii al client utilizzando respond_with come visto nell'esempio di utilizzo di OCamlHTTP nell'approccio imperativo.

Nell'approccio funzionale (sez. 2.2.1) invece le callback (siano esse semplici o avanzate) hanno visibilità di un canale di output connesso al client HTTP che ha inviato la richiesta. Al programmatore è lasciata quindi la libertà di scrivere direttamente su questo canale una risposta HTTP carattere per carattere. Viene però fortemente consigliato l'utilizzo delle funzioni del modulo `Http_daemon` atte alla creazione e all'invio di risposte al client.

Ognuna di esse richiede come ultimo argomento un valore di tipo canale di output e assume che tale canale sia connesso al client che ha inviato la richiesta.

[8] analogamente a quando accade utilizzando la funzione `Http_daemon.send_basic_headers`, si veda sez. 2.3

È necessario innanzitutto distinguere tra due grandi famiglie di funzioni del modulo Http_daemon atte a inviare risposte ai client:

- funzioni che inviano *frammenti* di risposta HTTP
- funzioni che inviano risposte HTTP *complete*

Le funzioni appartenenti alla prima famiglia hanno nomi prefissi dalla stringa `send_`, le funzioni appartenenti alla seconda famiglia hanno nomi prefissi dalla stringa `respond_`. Per l'invio di una singola risposta è necessario utilizzare *o* una serie di funzioni che inviano parti di risposta HTTP (questo approccio richiede una buona conoscenza del formato delle risposte HTTP) *oppure*, alternativamente, una singola funzione che invii una risposta HTTP completa.

Analizziamo ora brevemente le funzioni principali per la creazione e l'invio di risposte HTTP ai client.

Invio di frammenti di risposta HTTP

L'invio di una risposta HTTP a frammenti prevede l'invio ordinato dei seguenti elementi:

1. status line
2. sequenza di header (opzionale)
3. CRLF
4. corpo

Per l'**invio della status line** OCamlHTTP mette a disposizione del programmatore due funzioni: `send_status_line` e `send_basic_headers`; il prototipo delle due funzioni è identico:

```
val send_status_line:
   ?version:  Http_types.version ->
   ?code:  int -> ?status:  Http_types.status ->
   out_channel ->
      unit
```

```
val send_basic_headers:
   ?version:  Http_types.version ->
   ?code:  int -> ?status:  Http_types.status ->
   out_channel ->
      unit
```

Ognuna di queste funzioni permette al programmatore di specificare due delle tre componenti della status line: la versione HTTP e lo status code (quest'ultimo specificabile numericamente o utilizzando un formato più astratto). La terza componente della status line, la reason phrase, viene calcolata da OCamlHTTP in base allo status code indicato.

Oltre alla status line, send_basic_headers spedisce anche alcuni header che sono comunemente inviati dai server HTTP quali la data corrente (header `Date`) e l'identificazione del server HTTP (header `Server`) evitando al programmatore di doverli spedire in seguito.

Per l'**invio degli header** OCamlHTTP offre due funzioni: `send_header` e `send_headers` utilizzate rispettivamente per inviare un singolo header o una lista di essi.

```
val send_header:
   header:  string -> value:  string ->
   out_channel ->
      unit
```

```
val send_headers:
   headers:(string * string) list ->
   out_channel ->
      unit
```

Nel caso di invio di una lista di header, questi sono specificati passando a send_headers una lista di coppie nelle quali il primo elemento è il nome del header e il secondo il valore ad esso associato.

È possibile inviare una sequenza **CRLF** scrivendo direttamente la stringa "\r\n" sul canale di output oppure utilizzando la funzione `send_CRLF`:

```
val send_CRLF:
   out_channel ->
      unit
```

Per l'**invio** del corpo non sono fornite funzioni specifiche dato che questo non richiede formattazioni particolari, è sufficiente inviarlo sul canale di output utilizzando funzioni della libreria standard OCaml quali `output` o `output_string`.

Viene però messa a disposizione una funzione per l'invio di corpo letto da file: questa funzionalità è implementata nella funzione `send_file`.

```
type file_source = FileSrc of string | InChanSrc of in_channel
val send_file:  src:Http_types.file_source -> out_channel -> unit
```

Come risulta dal prototipo della funzione è possibile inviare il contenuto di un file acceduto mediante il suo nome all'interno del filesystem oppure indicando un canale di input[9].

È importante evidenziare che, affinché si possano mantenere le connessioni persistenti di HTTP 1.1, è necessario che il programmatore che invia una risposta HTTP utilizzando l'approccio *a frammenti* calcoli la dimensione del corpo della risposta e generi l'apposito header `Content-Length`.

Invio di risposte HTTP complete

Alternativamente all'invio di risposte frammento per frammento, OCamlHTTP permette di inviare risposte HTTP complete.

Utilizzando questo approccio si possono ignorare i dettagli riguardanti il formato di una risposta HTTP e si delega a OCamlHTTP la computazione della dimensione del corpo della risposta e la generazione del corrispondente header `Content-Length`. Come svantaggio è importante sottolineare che utilizzando questo approccio si perde la scalabilità per quanto riguarda l'occupazione di memoria delle risposte.

La funzione più generale disponibile per l'invio di risposte HTTP complete è `respond`:

```
val respond:
   ?body:string -> ?headers:(string * string) list ->
   ?version:Http_types.version ->
   ?code:int -> ?status:Http_types.status ->
   out_channel ->
      unit
```

Utilizzata senza specificare alcuno dei valori opzionali respond invia una risposta con codice di ritorno 200 (OK), header predefiniti (Date e Server), versione HTTP 1.1 e corpo vuoto. È possibile impostare il codice di ritorno, la versione HTTP e il corpo della risposta impostando opportunamente i parametri opzionali e specificare quali header aggiuntivi si vogliono inviare.

Esistono poi un insieme di funzioni utilizzate per facilitare l'invio di risposte che frequentemente vengono inviate dal server HTTP: `respond_not_found`, `respond_forbidden`, `respond_redirect`, `respond_error`.

```
val respond_not_found:
   url:string -> ?version:  Http_types.version ->
   out_channel ->
      unit
```

[9] il prototipo di questa funzione è piuttosto ambiguo

```
val respond_forbidden:
   url:string -> ?version:  Http_types.version ->
   out_channel ->
      unit
```

```
val respond_redirect:
   location:string -> ?body:string -> ?version:  Http_types.version ->
   ?code:  int -> ?status:  Http_types.redirection_status ->
   out_channel ->
      unit
```

```
val respond_error:
   ?body:string -> ?version:  Http_types.version ->
   ?code:  int -> ?status:  Http_types.error_status ->
   out_channel ->
      unit
```

Le prime tre funzioni inviano rispettivamente risposte con codici di ritorno 404, 403 e 302 indicanti rispettivamente che la risorsa richiesta non è stata trovata, che non è permesso accedervi e che è stata spostata altrove. respond_error può invece essere utilizzata per l'invio di errori generici.

È possibile assumere il comportamento di un semplice server HTTP che gestisca risorse statiche utilizzando la funzione `respond_file` che è in grado di inviare un file e appositi header al client o il listato del contenuto di una directory:

```
val respond_file:
   fname:string -> ?version:  Http_types.version ->
   out_channel ->
      unit
```

È infine possibile rispondere al client inviando oggetti di tipo request creati in precedenza utilizzando la funzione `respond_with`:

```
val respond_with:
   Http_types.response ->
   out_channel ->
      unit
```

2.4 URI escaping

Una delle attività più tediose nell'ambito della programmazione di applicazioni comunicanti utilizzando il protocollo HTTP è la gestione dell'escaping delle URI.

OCamlHTTP ovvia a questo problema effettuando l'URI escaping per conto del programmatore sulla request URI di ogni richiesta.

In pratica il programmatore che accede alla request URI o al path in esso contenuta o all'ambiente ricostruito da una richiesta riceva stringhe che sono già state sottoposte ad una *singola passata* di URI unescaping e non deve preoccuparsene.

2.5 Descrizione dei moduli

OCamlHTTP è strutturato internamente in 12 moduli, non tutti visibili nella API presentata al programmatore finale. Riportiamo una breve descrizione di ciascuno di essi.

Il grafo di dipendenza dei moduli è riportato in fig. 2.5.

Http_constants (non esposto) contiene la definizione di costanti utilizzate da altri moduli quali la versione HTTP predefinita e la server string.

Http_types contiene la definizione dei tipi utilizzati in OCamlHTTP. In particolare contiene le definizioni dei tipi rappresentanti: versioni HTTP, metodi delle richieste, stato di ritorno delle risposte, interfacce delle classi che implementano messaggi, richieste e risposte.

Http_common contiene parametri di configurazioni globali per OCamlHTTP quali le impostazioni di debugging e funzioni applicabili ai tipi definiti in Http_types quali parsing e pretty printing di versioni e metodi http e predicati applicabili ai codici di ritorno.

Http_misc (non esposto) sono qui implementate funzioni ausiliarie non strettamente correlate a OCamlHTTP, ma utilizzate per la sua implementazione

Http_parser e Http_parser_sanity (non esposti) in questi moduli sono implementate le funzioni di parsing di messaggi HTTP. Il modulo Http_parser_sanity contiene funzioni quali asserzioni sulla buona formazione di elementi di messaggi HTTP che possono essere utilizzate anche indipendentemente dalle funzioni di parsing definite in Http_parser.

Http_message, Http_request, Http_response contengono le implementazioni delle classi corrispondenti a messaggi, richieste e risposte HTTP.

Http_tcp_server e Http_threaded_tcp_server (non esposti) nel primo modulo sono implementati i diversi server TCP utilizzati per l'implementazione delle diverse modalità di gestione dei client di OCamlHTTP, nel secondo è implementata la versione con il supporto per il threading, è separata in un modulo separato per potere distribuire in una libreria separata la versione di OCamlHTTP con supporto per il threading.

Http_daemon implementa gli oggetti utilizzati per l'approccio imperativo alla scrittura di demoni HTTP.

2.6 Esempi di utilizzo

Riportiamo, a titolo di esempio, due diverse implementazioni di un servizio web che calcoli il fattoriale di un numero intero ricevuto in input.

La prima implementazione riportata (fig. 2.6) è stata realizzata seguendo l'approccio funzionale di OCamlHTTP. Per dimostrare le possibilità offerte dall'approccio multithreaded abbiamo implementato il calcolo del fattoriale invocando il servizio web ricorsivamente[10][11].

La seconda implementazione (fig. 2.7) è stata invece realizzata seguendo l'approccio imperativo. Il fattoriale viene calcolato senza invocazioni ricorsive del servizio web data l'impossibilità di gestire demoni multithreaded o multiprocesso nell'approccio imperativo di OCamlHTTP.

La libreria OCamlHTTP descritta in questa sezione, è stata utilizzata con successo per l'implementazione dei binding al protocollo HTTP dei servizi web implementati nell'ambito di questa tesi.

2.7 Implementazione

OCamlHTTP è composta da 12 moduli (fig. 2.5). Allo stato attuale l'implementazione consta di poco più di 3200 righe di codice OCaml comprensive di commenti, descrizione della API in formato ocamldoc ed esempi di utilizzo.

L'implementazione ha richiesto circa 250 ore/uomo di lavoro.

Lo sviluppo è stato condotto parallelamente all'utilizzo delle versioni preliminari di OCamlHTTP da parte degli altri servizi web implementati.

[10] La funzione *http_get* utilizzata non è parte di OCamlHTTP, ma è risultata necessaria data la natura ricorsiva dell'esempio (la callback deve infatti comportarsi anche da client HTTP)

[11] una implementazione della funzione *http_get* è disponibile nel modulo Http_client.Convenience, `http://www.ocaml-programming.de`

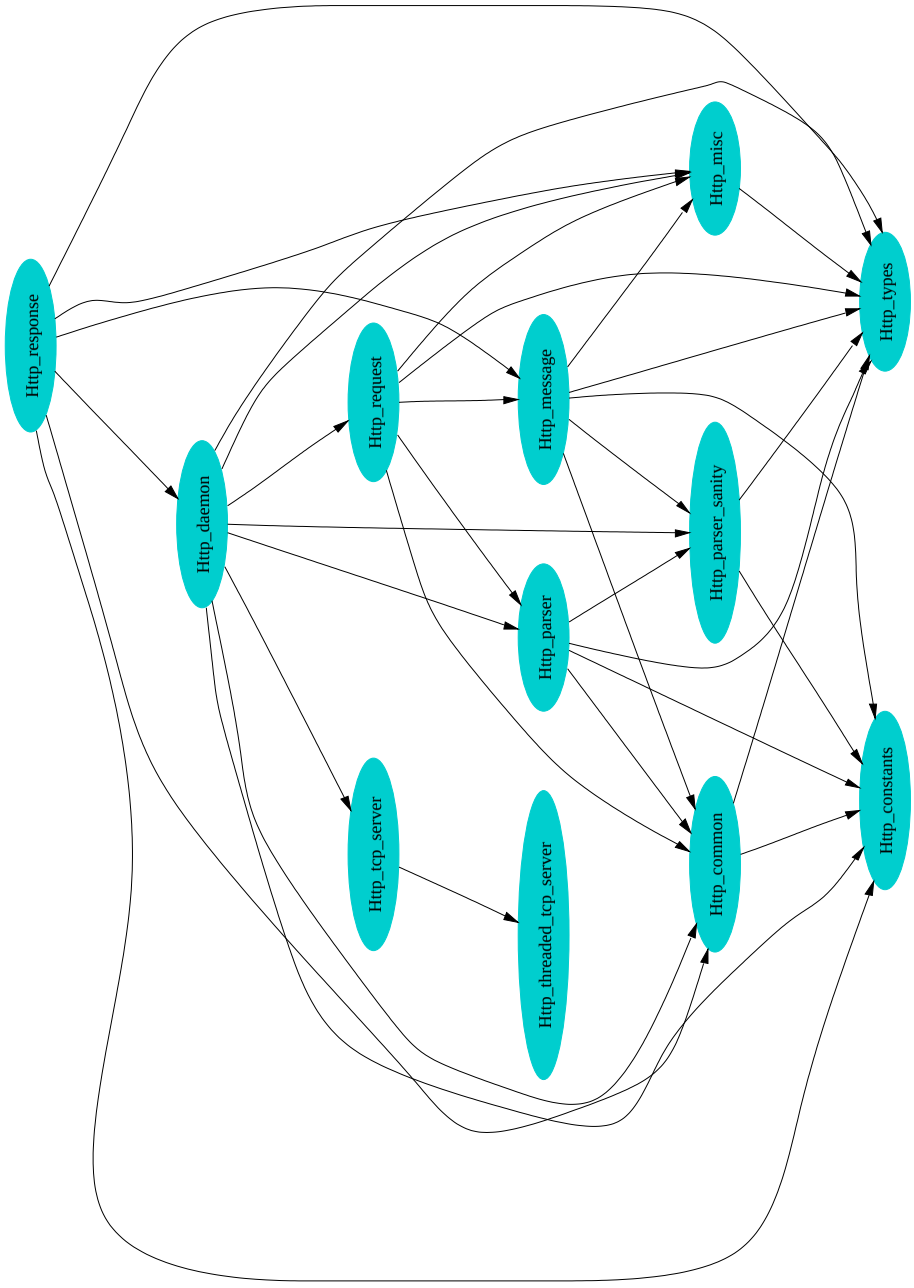


Figura 2.5: Dipendenza tra i moduli di OCamlHTTP

```
open Http_types;;
open Printf;;
let respond_fact x outchan =
  let response = new Http_response.response ~body:(string_of_int x) () in
  Http_daemon.respond_with response outchan
in
let callback (req: request) outchan =
  try
    (match req#path with
    | "/compute_fact" ->
        (match int_of_string (req#param "x") with
        | 0 -> respond_fact 1 outchan
        | x when x > 0 ->
            let fact =
              int_of_string (http_get
                (sprintf "http://localhost/compute_fact?x=%d" (x - 1)))
            in
            respond_fact fact outchan
        | x (* when x < 0 *) ->
            Http_daemon.respond_error
              ~body:"x must be greater than or equal to 0"
              outchan)
    | invalid_request ->
        Http_daemon.respond_error
          ~body:(sprintf "Request \"%s\" is not supported" invalid_request)
          outchan)
  with
  | Http_types.Param_not_found name ->
      Http_daemon.respond_error
        ~body:(sprintf "Parameter \"%s\" is required but not provided" name)
        outchan
  | Failure "int_of_string" ->
      Http_daemon.respond_error
        ~body:"Given value is not a valid integer number"
        outchan
in
Http_daemon.start' callback
```

Figura 2.6: Esempio di utilizzo di OCamlHTTP: approccio funzionale

```
open Http_types;;
open Printf;;
exception Invalid_fact_argument of int;;
let respond_fact x outchan =
  let response = new Http_response.response ~body:(string_of_int x) () in
  Http_daemon.respond_with response
in
let rec fact = function
  | 0 -> 1
  | x when x > 0 -> x * fact (x - 1)
  | x (* when x < 0 *) -> raise (Invalid_fact_argument x)
in
let build_error_response body =
  new Http_response.response ~status:(`Client_error `Bad_request) ~body ()
in
let daemon = new Http_daemon.daemon () in
while true do
  let (req, conn) = daemon#getRequest in
  try
    (match req#path with
    | "/compute_fact" ->
        let result = fact (int_of_string (req#param "x")) in
        let response =
          new Http_response.response ~body:(string_of_int result) ()
        in
        conn#respond_with response;
    | invalid_request ->
        conn#respond_with (build_error_response
          (sprintf "Request \"%s\" is not supported" invalid_request)));
    conn#close
  with
  | Http_types.Param_not_found name ->
      conn#respond_with (build_error_response
        (sprintf "Parameter \"%s\" is required but not provided" name));
      conn#close
  | Failure "int_of_string" ->
      conn#respond_with (build_error_response
        "Given value is not a valid integer number");
      conn#close
  | Invalid_fact_argument _ ->
      conn#respond_with (build_error_response
        "x must be greater than or equal to 0");
      conn#close
done
```

Figura 2.7: Esempio di utilizzo di OCamlHTTP: approccio imperativo

Capitolo 3

HTTP Getter: gestione della base documentaria

L'accessibilità delle informazioni è uno dei principi fondanti del progetto HELM. Per massimizzarla sono fondamentali sia il modello di distribuzione dei documenti contenuti all'interno della libreria che gli strumenti software disponibili per accedervi.

Il modello concettuale di distribuzione di HELM è stato quindi pensato principalmente per massimizzare le possibilità di collaborazione alla formazione della libreria. Altri scopi progettuali dello stesso sono stati: rendere possibili semplici meccanismi di resistenza ai guasti e di ottimizzazione delle prestazioni e minimizzare i requisiti di accesso ai documenti della libreria imposti agli utilizzatori.

L'applicazione software che implementa il modello di distribuzione e permette l'accesso alla libreria di HELM è stata nominata *getter*.

L'implementazione del getter finora utilizzata fu realizzata nel linguaggio di programmazione Perl[1]. Questa scelta implementativa ha influito negativamente sulla possibilità di manutenzione del codice al punto da rendere necessaria la sua riscrittura diverse volte.

In questo capitolo viene presentata una reimplementazione nel linguaggio OCaml del getter e vengono mostrati i benefici di questa scelta rispetto alla precedente implementazione.

[1] `http://www.perl.org`

3.1 Modello di distribuzione

L'approccio usuale di distribuzione di documenti nell'era del World Wide Web prevede la pubblicazione di risorse accessibili utilizzando il protocollo HTTP indirizzabili mediante il loro URL.

Come descritto in [Sac00] la scelta di URL per l'indirizzamento di documenti di conoscenza matematica risulta poco adatta. Questi particolari documenti infatti, a causa della loro interdipendenza e della necessità di avere a disposizione tutti i documenti tra loro dipendenti per potere effettuare con successo il type checking, richiedono caratteristiche di accessibilità particolari.

In particolare risultano evidenti due necessità. La prima (*replicazione*) richiede la possibilità di potere disporre di diverse copie di uno stesso documento, non necessariamente identiche, ma equivalenti dal punto di vista del tipaggio. La seconda (*trasparenza*) impone che il meccanismo di indirizzamento dei documenti non sia dipendente né dal server sul quali essi risiedono né tantomeno dal percorso di accesso a loro relativamente alla document root del server di residenza.

Il meccanismo di indirizzamento delle URL risulta quindi inadatto dato che permette di indirizzare una sola risorsa e contiene riferimenti sia al server di residenza che alla locazione fisica della stessa.

Meccanismo più adatto risulta invece quello degli URN abbinato ad una componente software in grado di effettuare internamente la consueta traduzione URN → URL e di restituire su richiesta documenti specificati tramite il loro URN. Un meccanismo di questo tipo rende l'utente finale ignaro della locazione fisica dei documenti e rende possibile la strutturazione dello spazio di URN in maniera più vicina alla struttura logica della base documentaria (piuttosto che alla sua struttura fisica).

Rendendo trasparente all'utente la provenienza di un documento si possono inoltre implementare facilmente meccanismi che supportino la *fault tolerance* quali la replicazione e risulta possibile la rilocazione di documenti senza causare il minimo disservizio.

Modelli di distribuzione molto simili a quello finora descritto sono stati già implementati, in particolare l'implementazione del getter si è fortemente ispirata al sistema di pacchettizzazione software della distribuzione *Debian*[2], noto con il nome di *APT - Advanced Package Tool*.

[2]http://www.debian.org

3.1.1 APT – Advanced Package Tool

Nel sistema APT ogni utente (ovvero ogni amministratore di macchine Debian) definisce una lista di server, specificandoli in ordine di preferenza, indicando ad APT quali siano le sorgenti da utilizzare per ottenere pacchetti software.

Impartendo poi il comando *update*, il sistema APT contatta ogni server specificato scaricando e salvando all'interno di una cache la lista dei pacchetti disponibili presso ognuno di essi. Ogni elemento di questa lista non è altro che una coppia che abbina al nome logico di un pacchetto software un URL che permetterà successivamente di scaricare il pacchetto in questione[3].

La concatenazione di queste liste (rispettando l'ordine dei server specificato dall'utente) forma una mappa che associa ad ogni nome logico del sistema di pacchettizzazione un URL di un file `.deb`, file contenente il pacchetto software nel formato utilizzato dalla distribuzione Debian.

Il sistema di pacchettizzazione fornisce poi all'utente le usuali operazioni quali la possibilità di installare nuovi pacchetti software e di aggiornare quelli presenti sul sistema. È importante sottolineare che l'utente di APT non ha mai a che fare con gli URL dei file .deb, ma solamente con il loro nome logico.

APT supporta inoltre una cache locale che permette di evitare di scaricare inutilmente lo stesso pacchetto software nel caso in cui questo sia già stato scaricato in precedenza.

Il modello di distribuzione di APT può quindi essere schematizzato come in fig. 3.1.

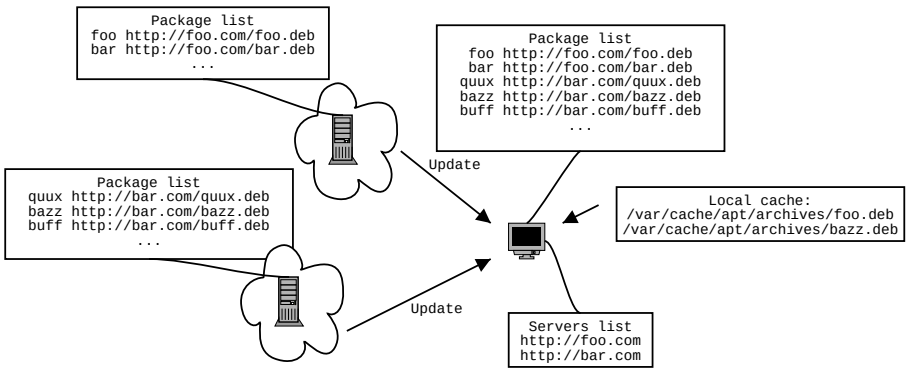


Figura 3.1: Modello di distribuzione di APT

[3]in realtà il sistema APT conserva all'interno di queste liste molte altre informazioni quali la versione del pacchetto, le sue dipendenze, ecc. Si è sorvolato su questo dettaglio in quanto di poco interesse per ciò che concerne il getter

3.1.2 HELM getter

Il modello di distribuzione di APT si è dimostrato adeguato alla distribuzione di documenti matematici all'interno di HELM ed è stato pertanto adottato adattandolo alle esigenze del caso.

La differenza principale consiste nelle modalità di utilizzo. prevede un unico client: l'applicazione software `apt-get` utilizzata dall'amministratore di sistema per la gestione dei pacchetti. Il getter prevede invece due diverse interfacce di accesso: un **modulo OCaml** ed un **servizio web**. Per il resto l'architettura è simile a quella vista per APT.

Qualsiasi utilizzatore del getter specifica una lista di server da utilizzare per creare una vista sulla base documentaria. Il getter dispone poi di diversi metodi per accedere alla base documentaria e di un metodo omonimo a quanto visto per APT (update) per ricreare la mappa che associa ad ogni URN di documenti della base documentaria un URL al quale è disponibile il documento. La mappa viene creata concatenando le liste scaricate da ogni server secondo l'ordine stabilito dalla lista dei server.

Una volta creata la mappa, è possibile utilizzare gli altri metodi del getter (descritti in sez. 3.5) per accedere alla base documentaria.

Il getter dispone inoltre di una cache locale che permette di evitare l'accesso a documenti remoti nel caso in cui questi siano già stati scaricati in precedenza.

Compressione È noto che la compressione di documenti XML utilizzando algoritmi basati su dizionario (quali ad esempio Lempel-Ziv, algoritmo utilizzato da `gzip`[4]) risulti molto efficace, permettendo di arrivare a rapporti di compressione fino a 1:20. Essendo la libreria di HELM interamente composta da documenti XML di dimensioni ragguardevoli, è risultato necessario implementare meccanismi di compressione e decompressione.

Il getter permette la gestione trasparente di documenti della base documentaria rispetto al loro stato di compressione.

Gli indici supportano un campo opzionale che indica lo stato di compressione di ogni oggetto reso disponibile presso un server. Il caso di documenti compressi è gestito trasparentemente dal getter: nel caso in cui venga richiesto l'accesso ad un documento compresso, questo viene decompresso prima di essere restituito all'utente.

[4] `ftp://alpha.gnu.org/gnu/gzip/`

3.2 Obiettivi progettuali

Gli obiettivi progettuali del getter sono stati definiti considerando non solamente le necessità architetturali dei componenti di HELM ma anche per superare i limiti evidenziati dallo studio delle precedenti implementazioni.

Gli obiettivi progettuali dello sviluppo del getter sono stati i seguenti:

trasparenza l'utente non deve avere alcuna necessità di conoscere la locazione fisica dei documenti, deve poter utilizzare la base documentaria conoscendo unicamente lo schema dei nomi logici (URN) della stessa

resistenza ai guasti deve essere possibile avere più repliche di uno stesso documento della base documentaria

rilocabilità deve essere possibile rilocare fisicamente un documento senza che tale modifica sia percepita dagli utenti

cancellabilità ogni autore di documenti deve avere la possibilità di rimuovere le copie in sue possesso degli stessi dalla base documentaria

estendibilità lo schema logico dei nomi della base documentaria deve essere estendibile, tale modifica deve potersi riflettere sul getter che deve essere quindi estendibile per supportare nuovi schemi di URN

3.3 Formato dei nomi logici (URN)

La scelta dei nomi logici deve essere consistente e intuitiva per gli utilizzatori finali. Dovendo inoltre il getter gestire diverse tipologie di documenti è risultato necessario strutturare rigorosamente lo spazio dei nomi logici.

Tutte le URN gestite dal getter sono URI[5] definite all'interno dello schema di URI "helm:". Lo schema generale di queste URN è il seguente[6]:

``helm:''*helm-uri-schema*``//''*helm-uri-specific-part*

[5] le URN sono un sottoinsieme delle URI, si veda [RFCc]

[6] attualmente in realtà le URN dei soli documenti della base documentaria *non* sono gestite secondo questo schema. Sono invece gestite secondo uno schema semplificato che prevede l'assenza del prefisso helm e dell'helm uri schema per motivi di compatibilità con il passato. La migrazione verso lo schema descritto è comunque programmata a breve

Il prefisso "helm:" è anteposto a tutte le URN gestite dal getter, l'*helm-uri-schema* identifica il sottoschema di URN utilizzato per l'interpretazione della URN, l'*helm-uri-specific-part* viene interpretato differentemente a seconda dell'helm uri schema.

L'helm uri schema è poi ulteriormente strutturato ed attualmente può assumere uno dei seguenti formati:

object:*logical-framework/software-system* per documenti matematici quali definizioni e teoremi definiti in un determinato sistema logico e provenienti dalla base documentaria di un certo sistema software (istanziazioni di questo helm uri schema sono ad esempio: `object:cic/coq` per la base documentaria del proof assistant Coq e `object:ml/nuprl` per la base documentaria del proof assistant NuPRL)

theory:*logical-framework/software-system* per identificare *teorie* ovvero raggruppamenti di documenti della base documentaria inerenti ad una argomento comune (esempio: la teoria dei limiti o la trigonometria). Analogamente al caso precedente sono specificati il sistema logico e l'applicazione software di provenienza. Un esempio di possibile istanziazione di questo schema di URI è `theory:/cic/coq`

rdf:*rdf-schema* per i documenti contenenti metadati relativi a documenti presenti nella base documentaria. La componente rdf-schema identifica il modello RDF (modello utilizzato per la formalizzazione dei metadati nel progetto HELM) di appartenenza del documento contenente i metadati. Un esempio di possibile istanziazione di questo schema di URI è `rdf:forward`

Diamo ora una breve descrizione delle helm uri specific part e della loro semantica nei diversi schemi di URI.

3.3.1 Schema *object:*

Lo scopo della parte locale delle URI nello schema object: è identificare un documento della base documentaria relativa al sistema logico ed alla applicazione software identificate dallo schema.

Tale parte locale rappresenta un percorso logico all'interno della base documentaria che permette di raggiungere le diverse parti di un documento. Non è lecito assumere l'esistenza di alcuna relazione tra questo percorso logico ed eventuali percorsi fisici (ad esempio all'interno di un file system) necessari a raggiungere il documento stesso.

Il percorso viene composto utilizzando la usuale sintassi delle URI che separa le componenti gerarchiche del percorso utilizzando il separatore "/". La risorsa finale identificata dal percorso può essere seguita da una o più estensioni responsabili di indicare quale *parte* del documento della base documentaria è contenuto nella risorsa indicata dalla URI.

A titolo di esempio riportiamo lo schema delle estensioni utilizzate per le URI dello schema `object:cic/coq`:

path`.(con|ind|var)[.(body|types)][.ann]`

La prima estensione è obbligatoria e può assumere uno dei seguenti valori:

.con per documenti che identificano *costanti* quali definizioni, teoremi e prove incomplete

.ind per documenti che identificano definizioni di tipi (co)induttivi

.var per variabili di sezione (concettualmente simili ad ambienti all'interno dei quali è possibile istanziare una prova)

La seconda estensione è presente solamente nel caso in cui la prima abbia assunto valore `.con`. Questa estensione può assumere valore `.body` per il corpo di costanti o `.types` per il loro tipo.

La terza estensione (`.ann`) è presente nel caso in cui il documento non rappresenti una componente della risorsa matematica vera, ma una annotazione a riguardo di uno di essi.

3.3.2 Schema *theory:*

Lo scopo della parte locale delle URI di questo schema è identificare raggruppamenti di documenti della base documentaria correlati tra loro secondo un certo criterio. Nella pratica le teorie sono utilizzate per raggruppare teoremi relativi ad una specifica branca della matematica.

La parte locale di queste URI è formata da un path logico (utilizzante l'usuale separatore "/") che identifica una specifica teoria nello spazio delle teorie.

3.3.3 Schema *rdf:*

Le URI di questo schema identificano metadati nel modello RDF rappresentati in XML relativi a documenti della base documentaria.

La parte locale di queste URI è definita identicamente alle URI dello schema `object:` con la differenza semantica che la risorsa identificata da queste URI

è un documento RDF contenente metadati che riguardano la corrispondente risorsa dello schema `object:`. Il modello RDF di appartenenza dei metadati così identificati è definito nel helm uri schema.

A titolo di esempio riportiamo la URI che rappresenta le dipendenze dell'oggetto corrispondente al numero naturale 0 (omettiamo lo schema di URI *helm:rdf:*):

www.cs.unibo.it/helm/rdf/forward//cic:/Coq/Init/Datatypes/nat.ind,1,1

3.4 Interfaccia

L'interfaccia del getter prevede metodi per l'accesso, la navigazione e la manutenzione della libreria[7].

3.4.1 Metodo *getxml*

Parametri : getxml supporta tre parametri.

Il parametro *uri* indica la URI del documento richiesto della base documentaria.

Il parametro *format* indica il formato richiesto di restituzione del documento, può assumere due valori: "normal" e "gz". Il primo valore indica che il documento deve essere restituito in formato non compresso, il secondo indica che il documento deve essere restituito in formato compresso utilizzando la compressione Lempel-Ziv, tipica dei tool gzip e gunzip. Nel caso questo valore sia selezionato il documento viene restituito aggiungendo alla risposta HTTP l'header "Content-Encoding: x-gzip". Il parametro format è opzionale, nel caso in cui non sia specificato il suo valore predefinito è normal.

Il parametro *patch_dtd* infine è obbligatorio e può assumere valori booleani (attualmente codificati nelle stringhe "yes" e "no" e relative modifiche di maiuscole/minuscole). Nel caso in cui patch_dtd sia impostato tutti i riferimenti a Document Type Definition esterne ([W3Ca]) nel documento restituito all'utente vengono rimpiazzate aggiungendo un livello di indirezione attraverso il getter. Se ad esempio il documento restituito facesse riferimento al DTD "cic.dtd", questo riferimento verrebbe

[7]la scelta dei nomi dei metodi risulta sotto molti aspetti inconsistente. Tale scelta è stata però imposta da motivazioni di compatibilità con il passato e con le altri componenti del progetto HELM

sostituito con "http://*getterUrl*/getdtd?uri=cic.dtd" nel quale getterUrl corrisponde alla URL del getter utilizzato

Richiesta HTTP GET :
/getxml?uri=URI&patch_dtd=(yes|no)[&format=(normal|gz)]

Valore di ritorno : il documento XML corrispondente alla URI specificata nel parametro uri

Descrizione : restituisce un documento della base documentaria corrispondente alla URI specificata nel parametro uri. Il getter si preoccupata della risoluzione da URI a URL e restituisce il documento eventualmente aggiungendo livelli di indirezione per l'accesso ai DTD ed eventualmente cambiando la codifica della risorsa per la spedizione della risposta HTTP

3.4.2 Metodo *getdtd*

Parametri : richiede un parametro obbligatorio *uri* che indica la URI del DTD richiesto. Supporta inoltre un parametro opzionale *patch_dtd*, analogo a quello visto per il metodo getxml. Se questo parametro è impostato tutti i riferimenti ad entità XML esterne presenti nel DTD richiesto vengono modificate aggiungendo un livello di indirezione attraverso il getter

Richiesta HTTP GET : */getdtd?uri=URI[&patch_dtd=(yes|no)]*

Valore di ritorno : il DTD corrispondente alla URI specificata nel parametro uri

Descrizione : restituisce il DTD presente all'interno della base documentaria corrispondente alla URI specificata nel parametro uri eventualmente modificato aggiungendo livelli di indirezione attraverso il getter per le entità esterne in base al valore del parametro patch_dtd

3.4.3 Metodo *getxslt*

Parametri : richiede un parametro obbligatorio *uri* che indica la URI del foglio di stile XSLT richiesto. Supporta inoltre un parametro opzionale *patch_dtd* analogo a quello visto per il metodo getxml. Se questo parametro è impostato i riferimenti ad altri fogli di stile effettuati utilizzando i costrutti ⟨xsl:import⟩ e ⟨xsl:include⟩ vengono modificati aggiungendo un livello di indirezione attraverso il getter

Richiesta HTTP GET : */getxslt?uri=URI[&patch_dtd=(yes|no)]*

Valore di ritorno : il foglio di stile XSLT corrispondente alla URI specificata nel parametro URI

Descrizione : restituisce il foglio di stile XSLT corrispondente alla URI specificata nel parametro URI eventualmente modificato aggiungendo un livello di indirezione attraverso il getter ai riferimenti a fogli di stile XSLT

3.4.4 Metodo *resolve*

Parametri : richiede e supporta il solo parametro *uri* indicante la URI della quale si richiede la risoluzione in URL

Richiesta HTTP GET : */resolve?uri=URI*

Valore di ritorno : la URL corrispondente alla URI specificata nel parametro uri. Più in dettaglio, la URL restituita è la URL del primo documento nella mappa del getter corrispondente a uri (ricordiamo infatti che la libreria di HELM supporta la replicazione di documenti: ad una singola URI possono corrispondere quindi più URL). L'ordine utilizzato per discriminare le URL corrisponde all'ordine dei server indicati nella configurazione del getter. Il formato di ritorno della URL è XML:

```
<url value="http://...." />
<unresolved />
```

Nel caso in cui la risoluzione vada a buon fine viene restituito l'elemento XML vuoto *url* con attributo *value* contenente la URL; nel caso in cui la risoluzione non vada a buon fine viene restituito l'elemento XML vuoto *unresolved*

Descrizione : implementa la risoluzione da URI a URL per i documenti della base documentaria e ritorna una rappresentazione XML della URL così ottenuta

3.4.5 Metodo *register*

Parametri : richiede due parametri: *uri* e *url* indicanti rispettivamente la URI da aggiungere alle mappe del getter e la corrispondente URL

Richiesta HTTP GET : */register?uri=URI&url=URL*

Valore di ritorno : non restituisce alcun valore significativo

Descrizione : modifica le mappe del getter aggiungendo ad esse una associazione URI → URL. Il tipo di URI viene automaticamente riconosciuto e l'associazione viene aggiunta solamente alla mappa opportuna (documenti CIC, fogli di stile XSLT, DTD, ecc.)

3.4.6 Metodo *update*

Parametri : nessuno

Richiesta HTTP GET : */update*

Valore di ritorno : un messaggio testuale che riporta traccia del processo di ricreazione delle mappe, eventuali errori ed altre informazioni utili (quali ad esempio l'accesso ad un server che non metta a disposizione alcun tipo di documenti)

Descrizione : ricrea le mappe interne del getter, ricostruendo così l'attuale *vista* sulla libreria. Il processo di ricostruzione prevede che il getter contatti tutti i server specificati nella sua configurazione, scarichi da ciascuno di essi gli indici dei documenti di cui dispongono e utilizzi questi indici per creare le associazioni URI → URL

3.4.7 Metodo *getalluris*

Parametri : nessuno

Richiesta HTTP GET : */getalluris*

Valore di ritorno : un indice testuale di tutti gli oggetti (schema *object:*) presenti all'interno della libreria

Descrizione : restituisce un indice di tutti gli oggetti contenuti nella libreria

3.4.8 Metodo *getallrdfuris*

Parametri : nessuno

Richiesta HTTP GET : */getallrdfuris*

Valore di ritorno : un indice testuale di tutti i metadati (schema *rdf:*) contenuti all'interno della libreria

Descrizione : restituisce un indice di tutti i metadati contenuti nella libreria

3.4.9 Metodo *ls*

Parametri : richiede un parametro *baseuri* indicante il path, relativo allo spazio delle URI della libreria, del quale si richiede di mostrare il contenuto. Richiede inoltre un parametro *format* indicante il formato richiesto di rappresentazione del contenuto. I valori supportati da questo parametro sono attualmente due: "txt" e "xml". Al primo corrisponde una rappresentazione testuale del contenuto, al secondo una rappresentazione XML.

Il formato di output testuale (fig. 3.2(a)) prevede una riga per ogni "elemento". Un elemento può essere una directory (ovvero una parte di libreria contenente altri elementi) oppure un oggetto. Nel caso in cui sia una directory viene riportato solamente il nome; nel caso in cui sia un oggetto vengono riportati il nome dell'oggetto e tre flag. Il primo di essi ha semantica booleana ed indica la presenza o meno di annotazioni sull'oggetto; il secondo può assumere valori "Yes", "No" e "Ann" indicanti, rispettivamente, la presenta di un file contenente il tipo dell'oggetto, la sua assenza e la sua presenza con relative annotazioni; il terzo ha semantica analoga al secondo, ma fa riferimento al file contenente il corpo dell'oggetto.

Il formato di output XML (fig. 3.2(b)) riporta le stesse informazioni codificate secondo il DTD di fig. 3.3. Per le directory viene utilizzato l'elemento ⟨section⟩ contenente il nome della directory; per gli oggetti l'elemento ⟨object⟩ contenente un elemento vuoto per ognuno dei flag applicabili agli oggetti. Ognuno di questi elementi ha un attributo obbligatorio indicante il valore del flag stesso.

Richiesta HTTP GET : */ls?baseuri=URI&format=(txt|xml)*

Valore di ritorno : una rappresentazione del contenuto di una parte della libreria indicata dal parametro baseuri

Descrizione : analogo ai comandi di *directory listing* tipicamente utilizzati per la gestione dei file system, ma applicato allo spazio dei nomi logici della base documentaria. Il formato baseuri specifica l'analogo di una directory per un file system. L'output del comando è l'analogo del contenuto di una directory per un file system. Per ogni elemento contenuto nella directory

```
dir, projections
object, Empty_set.ind, ⟨NO,YES,NO⟩
object, Empty_set_ind.con, ⟨NO,YES,YES⟩
object, Empty_set_rec.con, ⟨NO,YES,YES⟩
object, Empty_set_rect.con, ⟨NO,YES,YES⟩
object, bool.ind, ⟨NO,YES,NO⟩
object, bool_ind.con, ⟨NO,YES,YES⟩
object, bool_rec.con, ⟨NO,YES,YES⟩
object, bool_rect.con, ⟨NO,YES,YES⟩
object, fst.con, ⟨NO,YES,YES⟩
object, identity.ind, ⟨NO,YES,NO⟩
object, identity_ind.con, ⟨NO,YES,YES⟩
object, identity_rec.con, ⟨NO,YES,YES⟩
object, identity_rect.con, ⟨NO,YES,YES⟩
object, nat.ind, ⟨NO,YES,NO⟩
object, nat_ind.con, ⟨NO,YES,YES⟩
object, nat_rec.con, ⟨NO,YES,YES⟩
object, nat_rect.con, ⟨NO,YES,YES⟩
object, option.ind, ⟨NO,YES,NO⟩
object, option_ind.con, ⟨NO,YES,YES⟩
...
```

(a) Getter, metodo *ls*: formato testuale

```
⟨?xml version=1.0 encoding=ISO-8859-1?⟩
⟨!DOCTYPE ls SYSTEM http://.../ls.dtd⟩

⟨ls⟩
    ⟨section⟩projections⟨/section⟩
    ⟨object name=Empty_set.ind⟩
        ⟨ann value=NO /⟩
        ⟨types value=YES /⟩
        ⟨body value=NO /⟩
    ⟨/object⟩
    ⟨object name=Empty_set_ind.con⟩
        ⟨ann value=NO /⟩
        ⟨types value=YES /⟩
        ⟨body value=YES /⟩
    ⟨/object⟩
    ⟨object name=Empty_set_rec.con⟩
        ⟨ann value=NO /⟩
        ⟨types value=YES /⟩
        ⟨body value=YES /⟩
    ⟨/object⟩
    ⟨– ... –⟩
⟨/ls⟩
```

(b) Getter, metodo *ls*: formato XML

Figura 3.2: Getter, metodo *ls*: formati di output

vengono inoltre specificate informazioni aggiuntive quali la presenza di annotazioni, di informazioni di tipo, ecc.

3.4.10 Metodo *getempty*

Parametri : nessuno

Richiesta HTTP GET : */getempty*

Valore di ritorno : un documento XML *standalone*([W3Ca]) minimale, valido e ben formato

Descrizione : restituisce un documento XML minimale. Come esempio d'uso riportiamo l'esecuzione di trasformazioni XSLT che non richiedano alcun input

3.4.11 Metodo *help*

Parametri : nessuno

Richiesta HTTP GET : */help*

```
<!ELEMENT ls (section*|object*)>
<!ELEMENT section (#PCDATA)>
<!ELEMENT object (ann,types,body)>
<!ELEMENT ann EMPTY>
<!ATTLIST ann value (YES|NO) #REQUIRED>
<!ELEMENT types EMPTY>
<!ATTLIST types value (YES|NO|ANN) #REQUIRED>
<!ELEMENT body EMPTY>
<!ATTLIST body value (YES|NO|ANN) #REQUIRED>
```

Figura 3.3: Getter, metodo *ls*: DTD del formato di output XML

Valore di ritorno : un messaggio testuale contenente informazioni sul getter e una descrizione delle modalità di utilizzo dei suoi metodi

Descrizione : restituisce un messaggio di utilizzo e altre informazioni sulla corrente esecuzione del getter

3.5 Architettura

3.5.1 Base documentaria

La base documentaria gestita dal getter supporta attualmente diverse tipologie di documenti:

documenti XML costituenti la base documentaria del progetto HELM quali teoremi, definizioni e prove incomplete nei diversi sistemi logici supportati

fogli di stile XSLT utilizzati per la resa delle prove su web e internamente al proof assistant

DTD dei documenti XML utilizzati dalle varie componenti di HELM

3.5.2 Struttura interna

La struttura interna del getter può essere rappresentata schematicamente come in fig. 3.4.

Il modulo principale del getter implementa un server HTTP (utilizzando la libreria OCamlHTTP descritta in sez. 2) che offre una interfaccia, basata sul metodo GET di HTTP, ai metodi descritti in sez. 3.4.

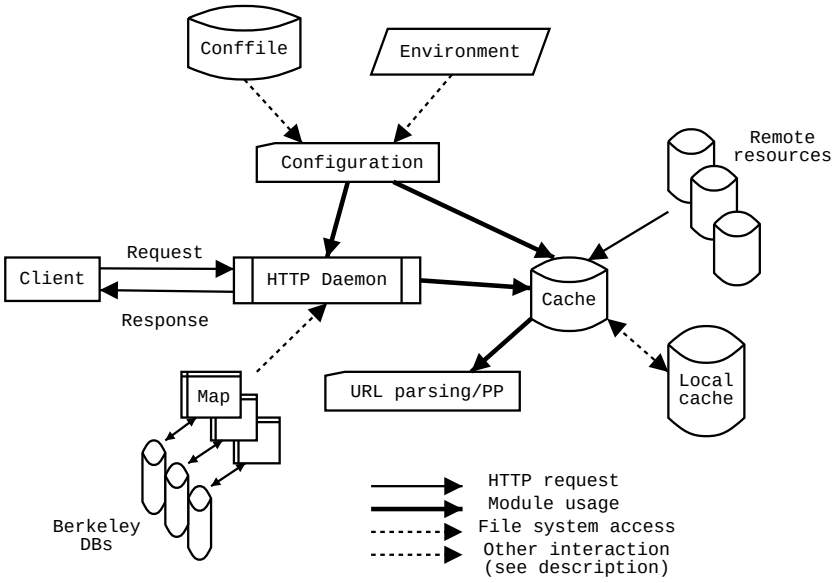


Figura 3.4: *getter*: architettura

Il demone HTTP (implementato nel modulo `Http_getter`) si preoccupa inoltre del parsing dei parametri passati dai client e dell'invio delle opportune risposte di errore HTTP nel caso in cui l'invocazione del getter non sia corretta. Come futura estensione è previsto che questo modulo si occupi anche di effettuare controlli di autenticazione in modo da potere restringere l'accesso ad alcuni metodi.

Il demone HTTP è multithreaded. Eseguendo i thread nello stesso spazio di memoria è così possibile modificare le mappe del getter rendendo le modifiche immediatamente visibili ai client che successivamente interpelleranno il getter.

Il demone HTTP ha accesso alle mappe (una per ogni tipologia di documenti presenti nella base documentaria[8]) di conversione da URN a URL. Queste mappe vengono utilizzate per implementare direttamente alcuni metodi dell'interfaccia del getter (ad esempio `resolve`) o per accedere alla cache.

Le mappe sono implementate fisicamente utilizzando database Berkeley DB ([OBS99]), particolarmente adatti per la gestione di archivi di coppie chiave–valore. L'accesso ai database fisici è interfacciato a livello OCaml dalla classe `map` definita nel modulo `Http_getter_map` la cui implementazione è thread safe grazie all'utilizzo di mutue esclusioni ove necessario. Questa caratteristica ha semplificato notevolmente l'implementazione della callback del demone HTTP.

[8]attualmente il getter gestisce quattro diverse mappe: una per gli oggetti CIC, una per gli oggetti NuPRL, una per i fogli di stile e una per i metadati RDF

Il getter accede alla cache ogni qual volta debba inviare un documento della base documentaria ad un client (ad esempio nel caso del metodo `getxml`). La cache offre un livello di astrazione sulla provenienza del documento. Nel caso in cui questo non sia presente nella cache, si preoccupa di scaricarlo dal server di origine e di aggiornare la cache locale, altrimenti restituisce al client la copia presente in essa.

Per chiarezza implementativa e comodità di gestione delle URL dei documenti, la cache non utilizza solamente una rappresentazione testuale delle stesse, ma anche un formato più astratto definito in `Http_getter_types`. Le funzioni di parsing e pretty printing di questo formato astratto sono disponibili all'interno di un apposito modulo.

I parametri di configurazione del getter, necessari sia al demone HTTP che alla cache, sono letti da un modulo apposito che offre un livello di astrazione sulla provenienza effettiva dei parametri di configurazione. Questi possono infatti provenire sia dal file di configurazione del getter che da variabili di ambiente.

3.5.3 Descrizione dei moduli

Il getter è strutturato internamente in 10 moduli il cui grafo di dipendenza è riportato in fig. 3.5.

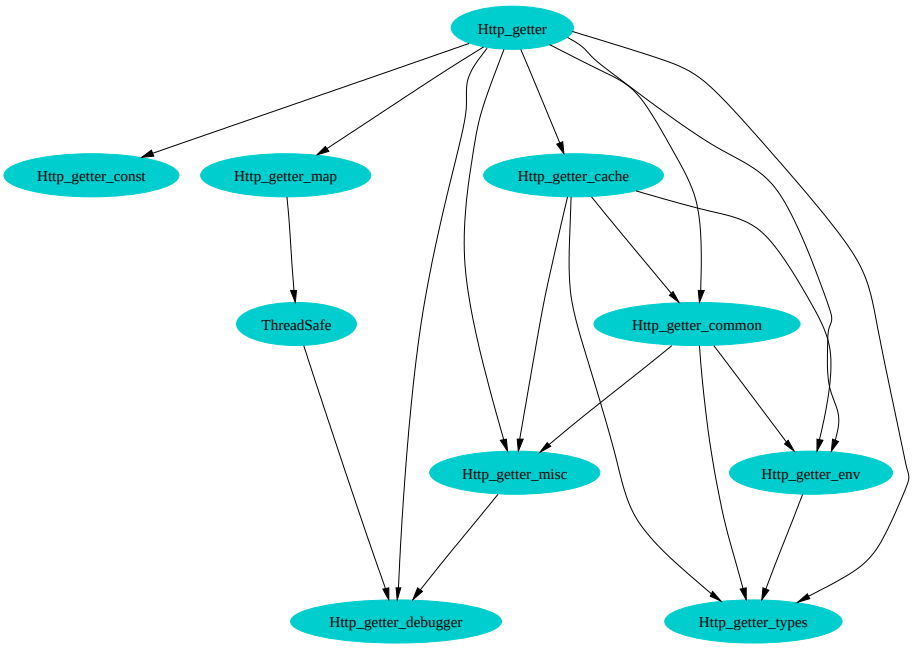


Figura 3.5: *getter*: dipendenza dei moduli

Riportiamo una breve descrizione di ciascuno di essi:

Http_getter_types contiene le definizioni di tipi utilizzati dal getter, in particolare contiene la formalizzazione delle URI nel tipo http_getter_uri

Http_getter_debugger implementa alcune funzioni di debugging e contiene parametri di configurazione globali relativi ad esse

Http_getter_misc contiene l'implementazione di funzioni non strettamente legate alle funzionalità del getter, ma utilizzate per la sua implementazione. Contiene inoltre interfacce OCaml a comandi di sistema utilizzati per implementare la cache

Http_getter_env racchiude le funzioni di accesso alla configurazione del getter e fornisce un livello di accesso ai parametri di configurazione che è indipendente dalla loro provenienza (file di configurazione o variabili di ambiente)

Http_getter_common contiene funzioni di uso comune negli altri moduli del getter quali: predicati applicabili alle URI, funzioni semplificate per l'invio delle risposte ai client e funzioni di parsing e pretty printing di URI

ThreadSafe implementa una classe che permette la derivazione di classi che abbiano la possibilità di definire metodi *thread safe* protetti da diverse tipologie di mutue esclusioni (in particolare permette di implementare metodi esclusivi e metodi utilizzabili secondo il modello lettori/scrittori)

Http_getter_cache implementa le funzioni utilizzate per l'invio di documenti ai client. Queste funzioni gestiscono in maniera trasparente la cache preoccupandosi di salvare in essa i documenti la prima volta che vengono acceduti e di preferire le copie locali in caso in cui queste esistano all'interno della cache

Http_getter_map implementa la classe utilizzata per mantenere le mappe URN $\rightarrow$ URL del getter (`map`). Istanze di questa classe offrono metodi thread safe che interfacciano i database Berkeley DB del getter presenti su disco

Http_getter_const contiene costanti utilizzate da altri moduli

Http_getter implementa il demone HTTP che offre, sotto forma di servizio web basato sul protocollo HTTP, le funzionalità del getter

3.6 Confronto con le precedenti implementazioni

L'implementazione precedente del getter fu realizzata (e riscritta da zero diverse volte) utilizzando il linguaggio di programmazione Perl. L'implementazione attuale OCaml presenta numerosi vantaggi rispetto a questa ed un solo svantaggio che, come vedremo, può considerarsi trascurabile e comunque risolvibile.

La **mantenibilità** dei sorgenti dell'attuale getter è di molto superiore a quella della precedente implementazione. Una valutazione di questo tipo non è quantificabile in unità di misura, ma riportiamo due esempi piuttosto significativi: la precedente implementazione Perl è stata riscritta praticamente da zero ben due volte nel tentativo di introdurre il supporto per nuovi metodi quali il metodo `ls`, richiedendo circa una settimana di lavoro per riscrittura. L'aggiunta del supporto per la mappa degli oggetti del sistema logico NuPRL all'attuale implementazione ha richiesto mezz'ora/uomo di lavoro. Il vantaggio di mantenibilità è probabilmente da imputarsi alla struttura fortemente modulare dell'attuale implementazione (la precedente constava di un unico file sorgente Perl), all'uso di un tipo di dato astratto per la manipolazione delle URL (l'implementazione Perl utilizzava semplici stringhe alle quali venivano periodicamente riapplicate le stesse espressioni regolari) a all'approccio funzionale tipico della programmazione nel linguaggio OCaml.

L'implementazione OCaml risulta molto più **scalabile** della precedente per quanto riguarda l'occupazione di memoria. La struttura interna della precedente prevedeva infatti l'intero contenimento in memoria dei documenti prima che questi fossero inviati all'utente.

Non abbiamo notato particolari differenze di **efficienza** tra le due implementazioni. Ciò è probabilmente imputabile al fatto che il codice critico dal punto di vista delle prestazioni risiede nell'implementazione dei Berkeley DB, implementazione condivisa dalle due implementazioni e realizzata esternamente nel linguaggio C.

La **mole** di codice sorgente delle due implementazioni è paragonabile: 939 righe Perl contro 1116 righe OCaml; considerando l'overhead aggiunto dalla maggiore modularizzazione, la mole di sorgenti OCaml risulta più che soddisfacente.

La **robustezza** della nuova implementazione è decisamente migliore della precedente in quanto effettua controlli più severi sulla correttezza degli argomenti passati dai client.

L'unico svantaggio della attuale implementazione rispetto alla precedente è

presente durante l'esecuzione del metodo `update`. In questa fase infatti l'attuale implementazione conserva interamente in memoria gli indici ottenuti dai server per potere effettuare controlli su di essi in maniera implementativamente semplice ed elegante. La precedente implementazione non soffriva di questa limitazione. Il problema risulta comunque trascurabile data l'esigua occupazione di memoria degli indici stessi.

3.7 Implementazione

Il Getter è composto da 10 moduli (fig. 3.5).

L'implementazione consta di poco più di 1100 righe di codice OCaml comprensive di commenti e di alcuni tool aggiuntivi (ad esempio un tool utilizzato per visionare il contenuto di un database Berkeley DB).

La realizzazione ha richiesto circa 80 ore/uomo di lavoro.

Capitolo 4

UWOBO: processore di *catene* XSLT

I documenti della libreria di HELM sono conservati in formati XML che variano a seconda del sistema logico di appartenenza dei documenti stessi.

La versatilità del formato XML permette di scrivere facilmente applicazioni che elaborino questi documenti. Affinché però questi documenti possano essere mostrati ad un utente finale risulta necessaria una loro elaborazione che li trasformi in un formato più fruibile per l'utente.

L'architettura tipica di applicazioni document centric basate su formato XML prevede la presenza di componenti software che siano in grado di elaborare direttamente rappresentazioni XML, abbinate ad una o più componenti che trasformino queste rappresentazioni in formati fruibili dall'utente finale (ad esempio HTML). Per questo tipo di trasformazioni viene spesso utilizzato il linguaggio XSLT ([W3Ci]) appositamente progettato dal W3C per descrivere trasformazioni di documenti XML.

I due client principali del progetto HELM, il sito web ed il proof assistant, applicano trasformazioni XSLT ai documenti XML della base documentaria per trasformarli in formati comprensibili ad un browser (per il sito web) o al widget GTK di resa di MathML (per il proof assistant).

La complessità delle trasformazioni necessarie ha imposto una strutturazione dei fogli di stile tale per cui la resa finale di un documento si ottenga solamente dopo l'applicazione di una *catena* di fogli di stile XSLT (4.4) piuttosto che di un singolo di essi.

I processori XSLT attualmente disponibili non supportano applicazioni di fogli di stile a catena. In questa tesi presentiamo quindi una applicazione software, basata su un processore XSLT esterno, in grado di eseguire questo tipo di

trasformazioni. L'applicazione software presentata prende il nome di UWOBO[1] ed offre all'utilizzatore una interfaccia di tipo servizio web basato sul protocollo HTTP.

4.1 Interfaccia

L'interfaccia di UWOBO prevede un insieme di metodi utilizzati per la gestione dei fogli di stile, un metodo per l'applicazione di una catena di stylesheet ed alcuni metodi ausiliari.

4.1.1 Metodo *add*

Parametri : una lista (di lunghezza ≥ 1) di *binding*; ogni binding è una stringa `chiave ‘‘,’’ uri`

Richiesta HTTP GET : */add?bind=key,uri[&bind=key,uri[&...]]*

Valore di ritorno : un messaggio di log relativo alla elaborazione di ogni singolo stylesheet

Descrizione : aggiunge agli stylesheet XSLT gestiti da UWOBO una lista di stylesheet; ognuno di essi è descritto indicando una URL utilizzabile per accedervi e una chiave utilizzata successivamente per referenziarlo. Non possono essere utilizzate chiavi già in uso dagli stylesheet attualmente gestiti da UWOBO

4.1.2 Metodo *remove*

Parametri : una lista di chiavi (eventualmente vuota) che referenzino stylesheet attualmente caricati da UWOBO

Richiesta HTTP GET : */remove[?keys=key1,key2,...]*

Valore di ritorno : un messaggio di log relativo alla rimozione di ogni singolo stylesheet

Descrizione : rimuove dall'insieme di stylesheet attualmente gestiti da UWOBO ogni stylesheet che abbia come chiave una chiave passata come

[1] una implementazione Java precedente della stessa applicazione fu realizzata da Luca Padovani durante un periodo di soggiorno in Western Ontario. Il nome UWOBO è la contrazione di University of Western Ontario university of BOlogna

argomento. Nel caso in cui la lista di chiavi sia vuota tutti gli stylesheet vengono rimossi

4.1.3 Metodo *reload*

Parametri : una lista di chiavi (eventualmente vuota) che referenzino stylesheet attualmente caricati da UWOBO

Richiesta HTTP GET : */reload[?keys=key1,key2,...]*

Valore di ritorno : un messagio di log relativo al ricaricamento di ogni singolo stylesheet

Descrizione : segnala ad UWOBO di *ricaricare*[2] una lista di stylesheet che erano stati precedentemente caricati. Le chiavi devono referenziare stylesheet attualmente caricati da UWOBO. Nel caso in cui la lista di chiavi sia vuota tutti gli stylesheet vengono ricaricati

4.1.4 Metodo *list*

Parametri : nessuno

Richiesta HTTP GET : */list*

Valore di ritorno : una rappresentazione testuale dello stato degli stylesheet attualmente caricati da UWOBO comprendente l'URL dal quale sono stati scaricati, la chiave ad essi associata e altre informazioni

Descrizione : restituisce la lista degli stylesheet attualmente caricati da UWOBO e alcune informazioni ad essi associate

4.1.5 Metodo *apply*

Parametri : richiede obbligatoriamente un parametro *xmluri* indicante la URL del documento XML al quale applicare la catena di trasformazioni XSLT e un parametro *keys* contenente una lista di chiavi di fogli di stile (separate da virgola) referenzianti gli stylesheet che compongono la catena, l'ordine di questa lista è significativo: il primo stylesheet di essa è il primo ad essere applicato. È possibile inoltre specificare parametri opzionali quali i parametri da passare ai fogli di stile (si veda sez. 4.4.1), questi possono

[2] ovvero di scaricare e rielaborare ogni singolo stylesheet

essere specificati nel formato *param.name=value* per specificare parametri *globali* che verranno passati a tutti gli stylesheet oppure nel formato *param.key.name=value* per specificare parametri che verranno passati solamente allo stylesheet referenziato dalla chiave *key*. È infine possibile indicare le proprietà di output che saranno utilizzate nell'ultima trasformazione della catena per generare il risultato finale (si veda sez. 4.4.2) utilizzando la sintassi *prop.name[=value]*. Il valore associato alla proprietà è opzionale in quanto non tutte le proprietà richiedono un valore. Per la lista delle proprietà di output supportate si veda sez. 4.4.2.

Richiesta HTTP GET :

/apply?xmluri=uri&keys=key1,key2,...
[&param.name=value[&param.name=value[&...]]]
[&param.key.name=value[&param.key.name=value[&...]]]
[&prop.name[=value][&prop.name[=value][&...]]]

Valore di ritorno : l'esito dell'applicazione della catena di stylesheet al documento di input

Descrizione : applica una catena di stylesheet specificati ordinatamente utilizzando il parametro keys al documento specificato dalla URL xmluri e restituisce il document finale della catena di trasformazioni. L'applicazione degli stylesheet può essere parametrica nei parametri globali (passati a tutti gli stylesheet) e nei parametri passati specificatamente ad alcuni stylesheet della catena. L'output finale della trasformazione può essere modificato dalle proprietà di output richieste.

4.1.6 Metodo *help*

Parametri : nessuno

Richiesta HTTP GET : */help*

Valore di ritorno : un messaggio testuale contenente informazioni su UWOBO ed una descrizione delle modalità di utilizzo dei suoi metodi

Descrizione : restituisce un messaggio di utilizzo e altre informazioni sulla corrente esecuzione di UWOBO

4.2 Architettura

L'architettura interna di UWOBO può essere schematizzata come in fig. 4.1.

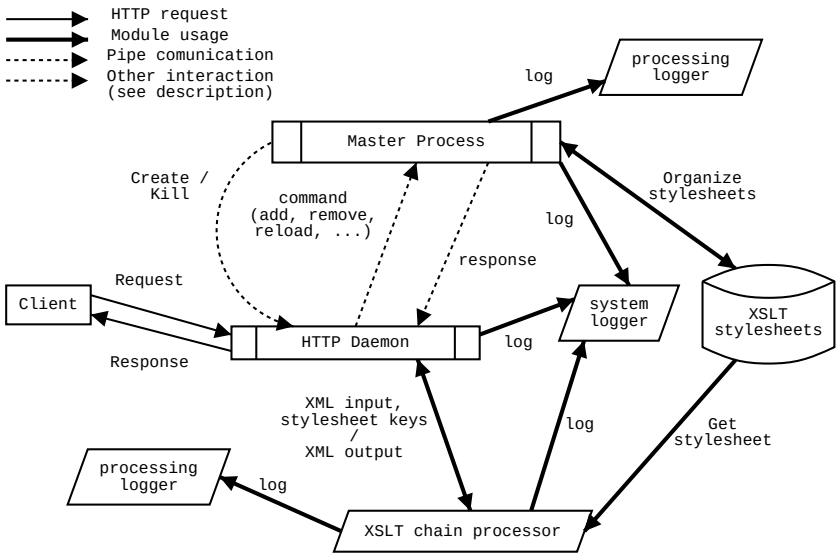


Figura 4.1: UWOBO: architettura

L'interfaccia di utilizzo di UWOBO è un servizio web basato su protocollo HTTP, il punto principale di accesso ad UWOBO risulta quindi essere (analogamente a quanto visto per il getter) un demone HTTP.

Problemi di rientranza del processore XSLT utilizzato (si veda sez. 4.5) hanno forzato la scelta di una architettura multiprocesso rispetto ad una multithread che avrebbe semplificato la gestione dei fogli di stile.

L'architettura multiprocesso scelta prevede l'esistenza di un processo principale, chiamato *master process*, detentore dell'archivio degli stylesheet attualmente caricati.

Il master process si preoccupa di istanziare un processo che implementi il demone HTTP responsabile di dialogare con i client di UWOBO e di mantenere due canali di comunicazione (implementati utilizzando *pipe* POSIX) con questo processo: uno chiamato *command pipe* e uno chiamato *response pipe*. Questi due canali di comunicazione vengono utilizzati per la gestione di tutte le operazioni che riguardano l'organizzazione degli stylesheet.

Ogni qual volta l'insieme degli stylesheet attivo viene cambiato è necessario che le modifiche siano effettuati nello spazio di memoria del master process e che il processo che implementa il demone HTTP sia rieseguito in modo da avere

visibilità delle modifiche avvenute. Per maggiori dettagli sulla gestione degli stylesheet si veda sez. 4.3.

Per la gestione di richieste che non richiedano riorganizzazione degli stylesheet non vi è comunicazione tra i processi che gestiscono le richieste ed il master process.

Per la gestione delle richieste di tipo *apply* i processi che gestiscono le richiesta hanno accesso al processore di catene di stylesheet di UWOBO (XSLT *chain processor*) che si preoccupa di applicare gli stylesheet e di impostare parametri e proprietà di output in base alle richieste dell'utente. Questo processore ha accesso in sola lettura all'archivio degli stylesheet di UWOBO. Il chain processor è basato sul processore XSLT *libxslt* ([Vei03b]), sul parser XML *libxml* ([Vei03a]), sull'implementazione DOM libgdome2 ([CP02a], [CP02b]) e sui rispettivi binding per il linguaggio OCaml di tali componenti software.

Sono infine presenti due diversi tipi di *logger*. Uno di essi viene utilizzato per la registrazione di messaggi di sistema di interesse globale (quali la ricezione di connessioni da parte di client o errori interni di comunicazione tra il master process e il demone HTTP), questo logger prende il nome di *system logger*. L'altro logger prende il nome di *processing logger* e viene utilizzato per raccogliere messaggi di elaborazioni eseguite su un insieme di elementi (ad esempio un insieme di stylesheet) e per inviarli successivamente al client. Il processing logger supporta diversi formati testuali di output ed effettua automaticamente escaping di caratteri non ammessi nel formato utilizzato.

4.2.1 Descrizione dei moduli

L'implementazione di UWOBO è divisa in cinque moduli distinti dei quali riportiamo una breve descrizione. Il grafo di dipendenza dei moduli di UWOBO è riportato in fig. 4.2.

Uwobo_common contiene le definizioni di costanti di uso comune negli altri moduli e di funzioni di comodo utilizzate per rispondere ai client e per accedere all'elenco delle proprietà di output supportate

Uwobo_logger contiene l'implementazione delle classi corrispondenti ai due logger utilizzati da UWOBO

Uwobo_styles contiene l'implementazione della classe `styles` responsabile della gestione dei fogli di stile all'interno del master process

Uwobo_engine contiene l'implementazione del processore di catene di stylesheet di UWOBO

Uwobo è il main program di UWOBO, sono qui implementati il master process, il demone HTTP e le comunicazioni tra loro

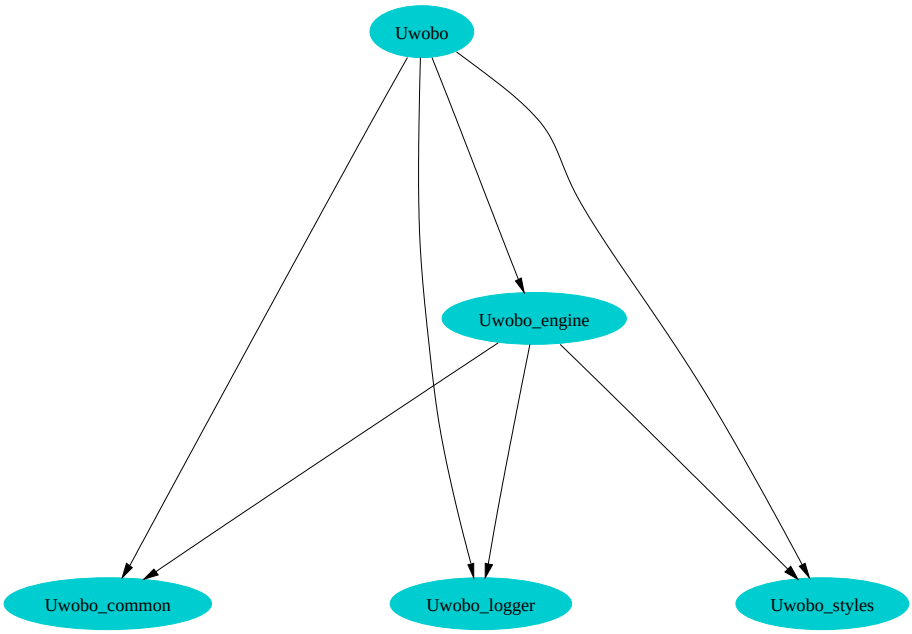


Figura 4.2: UWOBO: dipendenza dei moduli

4.3 Gestione dei fogli di stile

La scelta dell'architettura multiprocesso impone che non vi sia condivisione di memoria implicita tra i diversi processi che compongono UWOBO.

Questo vincolo risulta particolarmente problematico per quanto riguarda la gestione degli stylesheet. La richiesta di modifica all'insieme degli stylesheet disponibili infatti viene notificata ai processi creati dal demone HTTP per la gestione delle richieste, ma l'insieme degli stylesheet disponibili è gestito dal master process e risiede quindi nel suo spazio di memoria.

Risulta quindi necessario implementare le seguenti funzionalità:

- invio di comandi di gestione degli stylesheet dai processi che gestiscono le richieste HTTP al master process
- invio di feedback dal master process ai processi che gestiscono le richieste HTTP (in modo che questi possano rispondere al client a riguardo dei risultati delle sue richieste)

- aggiornamento dell'insieme dei fogli di stile percepito dai processi che gestiscono le richieste HTTP

Queste funzionalità sono implementate all'interno di UWOBO utilizzando le due pipe viste in precedenza e i meccanismi offerti dallo standard POSIX per la creazione e la terminazione di processi.

La *command pipe* viene utilizzata dai processi che gestiscono le richieste HTTP per inviare comandi di gestione degli stylesheet al master process, la *response pipe* viene utilizzata invece da quest'ultimo per inviare ai primi le risposte. Queste risposte contengono principalmente messaggi di log riguardo all'esecuzione del comando richiesto in modo che i processi di gestione delle richieste HTTP possano a loro volta inviare feedback significativo ai client.

Ogni qual volta l'insieme degli stylesheet viene modificato all'interno del master process, questi si preoccupa di terminare (utilizzando la chiamata di sistema POSIX `kill`) il processo che implementa il demone HTTP di UWOBO e di crearne uno nuovo (utilizzando la chiamata di sistema POSIX `fork`) che avrà quindi visibilità del nuovo stato degli stylesheet (grazie alla copia dello spazio di memoria implementata da fork). I processi che verranno successivamente creati per la gestione delle richieste HTTP "erediteranno" questo nuovo stato dal demone HTTP.

Per maggiore chiarezza riportiamo un esempio di modifica dello stato degli stylesheet.

Esempio: Un client contatta l'interfaccia HTTP di UWOBO inviando un comando che imponga la modifica dello stato attuale dei fogli di stile (ad esempio /add). Il processo creato dal demone HTTP[3] per la gestione della richiesta del client invia al master process un comando sulla command pipe corrispondente alla richiesta del client (nel nostro esempio il comando indicherà al master process che il client ha richiesto il caricamente di un insieme di nuovi stylesheet) e si pone in attesa di una risposta sulla response pipe. Il master process appena ricevuto il comando lo processa ed esegue le modifiche necessarie sull'insieme degli stylesheet attualmente caricati (nel nostro esempio scaricherà i nuovi stylesheet, li processera e li aggiungerà alla lista degli stylesheet caricati). Una volta terminata questa operazione invia sulla response pipe una risposta corrispondente al log di elaborazione del comando. Il processo che sta gestendo la richiesta del client riceve questa risposta e la inoltra al client terminando poi la connessione. Successivamente all'invio della risposta sulla response pipe il master process termina il processo che implementa il demone HTTP e ne crea uno

[3]il demone HTTP è implementato utilizzando OCamlHTTP in modalità 'Fork

nuovo in modo che le successive richieste siano gestite da processi che abbiano visibilità delle modifiche apportate alla lista di stylesheet[4].

4.4 Catene di fogli di stile

Schematizziamo in questa sezione l'idea della applicazione di una catena di fogli di stile ad un singolo documento utilizzando il paradigma di programmazione funzionale.

Consideriamo innanzitutto i tipi correlati alla applicazione di fogli di stile XSLT a documenti XML. Possiamo definire due tipi astratti: uno per i documenti XML e uno per i fogli di stile XSLT[5].

```
type stylesheet
```

```
type document
```

Possiamo ora immaginare di avere a disposizione una funzione, resa disponibile dal processore XSLT utilizzato, che applichi fogli di stile XSLT a documenti XML generando nuovi documenti XML. Sia questa funzione *apply*, il suo tipo sarà:

```
val apply:  document -> stylesheet -> document
```

il primo argomento è il documento XML soggetto della trasformazione, il secondo il foglio di stile XSLT descrivente la trasformazione, il valore di ritorno è il documento XML generato dalla trasformazione.

L'idea della applicazione di una catena di stylesheet consiste nell'applicare il primo foglio di stile al documento XML originale, il secondo foglio di stile al risultato di questa trasformazione, ecc. fino all'ottenimento del documento finale come risultato della trasformazione associata all'ultimo foglio di stile della catena.

Il tipo associato alla catena di stylesheet è naturalmente il tipo "lista di stylesheet":

```
stylesheet list
```

[4]tali modifiche non sarebbero altrimenti visibili in quanto, essendo l'architettura multiprocesso, non c'è condivisione di memoria implicita tra processi

[5]nella implementazione di UWOBO questi tipi sono vincolati dalla librerie utilizzate: per i fogli di stile XSLT, gdome2-xslt definisce il tipo `I_gdome_xslt.processed_stylesheet`, analogamente per i documenti XML, gdome2 definisce il tipo `Gdome.document`

L'applicazione di una catena di stylesheet è ora facilmente formalizzabile utilizzando una funzione "classica" del paradigma di programmazione funzionale nota come *fold_left*, la cui definizione è la seguente:

```
1 let rec fold_left f acc = function
2   | [] -> acc
3   | hd::tl -> fold_left f (f acc hd) tl
```

Utilizzando questa funzione, apply e le definizioni che abbiamo dato, l'applicazione di una catena di fogli di stile è implementata dalla funzione seguente *chain_apply* della quale riportiamo anche il prototipo:

```
val chain_apply:  document -> stylesheet list -> document

let chain_apply = fold_left apply
```

Nell'implementazione reale di UWOBO le operazioni da svolgere sono più complesse[6], ma il modello concettuale della applicazione di catene di stylesheet risulta immutato rispetto a quanto descritto.

4.4.1 Fogli di stile *parametrici*

La specifica del linguaggio XSLT ([W3Ci]) permette la definizione di template parametrici in uno o più parametri.

L'istanziazione di questi parametri può avvenire all'atto dell'invocazione del template (tipicamente effettuata mediante il costrutto ⟨`xsl:call-template`⟩) specificando il valore dei parametri utilizzando il costrutto ⟨`xsl:with-param`⟩.

Alternativamente alcuni processori XSLT permettono di specificare esternamente, all'atto cioè dell'utilizzo del processore, i valori assunti dai parametri dei vari template. Il processore `libxslt` utilizzato per l'implementazione di UWOBO offre questa funzionalità.

In particolare il binding OCaml di libxslt utilizzato (gdome2-xslt) offre per l'esecuzione di trasformazioni XSLT una funzione avente il seguente prototipo:

[6] La complessità addizionale deriva dal supporto di parametri e proprietà di output. Il supporto per i parametri impone che la funzione apply richieda anche un argomento aggiuntivo contenente i parametri. Tale parametro deve essere opportunamente istanziato. Il supporto per le proprietà di output impone che l'ultimo foglio di stile della catena sia trattato diversamente dagli altri. Viene infatti valutata la presenza di un elemento ⟨xsl:output⟩ al suo interno da utilizzare per ottenere i valori di default delle proprietà di output. L'ultimo foglio di stile devo poi essere modificato in base alle proprietà di output specificate dall'utente prima di essere di essere utilizzato per la serializzazione del risultato finale della catena.

```
val applyStylesheet: source: Gdome.document ->
stylesheet:I_gdome_xslt.processed_stylesheet -> params:(string * string) list
-> Gdome.document
```

Il parametro *params* è una lista di coppie *nome*, *valore*, che specificano parametri da passare ai template utilizzati durante la trasformazione.

UWOBO si preoccupa quindi internamente di calcolare l'insieme di parametri da passare ad ogni foglio di stile (i parametri globali vengono passati a tutti gli stylesheet coinvolti nella trasformazione, mentre i parametri specifici vengono passati solamente agli stylesheet richiesti dall'utente) e di utilizzarli durante l'applicazione della catena di trasformazioni XSLT vista in precedenza.

4.4.2 Proprietà di output

La specifica XSLT ([W3Ci]) permette all'autore di fogli di stile di indicare non solamente quali trasformazioni apportare al documento XML di input ma anche di impartire al processore XSLT direttive per ciò che concerne il formato di output finale della trasformazione.

In particolare il costrutto XSLT ⟨`xsl:output`⟩ supporta un insieme di proprietà (dette *proprietà di output*) che influenzano tale formato. UWOBO supporta la totalità delle proprietà di output definite nella specifica XSLT; per una descrizione di queste proprietà si veda tab. 4.4.2.

Nel caso della applicazione di catene di fogli di stile risulta evidente che le proprietà di output sono significative solamente per l'ultima trasformazione in quanto è l'unica che effettivamente genera output per l'utente.

UWOBO ammette diverse provenienze per le impostazioni relative alle proprietà di output. In particolare vengono considerate, in questo ordine di importanza:

1. le proprietà passate al metodo *apply*
2. i valori definiti dall'elemento ⟨xsl:output⟩ dell'ultimo foglio di stile di ogni catena
3. i valori predefiniti del processore XSLT in uso

L'ordine di importanza fa sì che, se nessuna proprietà è specificata all'interno dell'ultimo foglio di stile o come argomento del metodo apply, i valori predefiniti del processore XSLT siano utilizzati. È lasciata poi libertà all'utente di ridefinire tali valori agendo sull'ultimo foglio di stile o sui parametri del metodo apply.

Proprietà	**Descrizione**
cdata-section-elements	richiede come parametro una lista di nomi di elementi il cui contenuto sarà racchiuso all'interno di sezioni CDATA
doctype-public	specifica l'identificatore pubblico da utilizzare nella dichiarazione di tipo di documento
doctype-system	specifica l'identificatore di sistema da utilizzare nella dichiarazione di tipo di documento
encoding	specifica la codifica di caratteri del documento generato, questa verrà anche menzionata nella dichiarazione XML dello stesso
indent	richiede un parametro booleano che indica se generare spazi addizionali per l'indentazione del documento generato
media-type	specifica il *media-type* del documento generato. L'argomento non deve includere la codifica dei caratteri dato che verrà utilizzata quella specificata dalla proprietà *encoding*
method	richiede un parametro che può assumere uno dei seguenti valori: *xml*, *html*, *text*. Il parametro indica la *modalità* di generazione di output del processore XSLT (per una descrizione più dettagliata delle modalità di output si rimanda a [W3Ci])
omit-xml-declaration	richiede un parametro booleano che indica al processore XSLT se omettere la dichiarazione XML nel documento generato
standalone	richiede un parametro booleano che indica al processore se generare la dichiarazione *standalone document* o meno
version	richiede un parametro che indica la versione da utilizzare della modalità di output indicata dalla proprietà *method*

Tabella 4.1: UWOBO: proprietà di output

I valori predefiniti del processore XSLT vengono ovviamente utilizzati dal processore stesso senza nessuna necessità di interventi da parte dell'implementazione UWOBO, così come i valori definiti nell'ultimo foglio di stile della catena (assumendo che il processore XSLT sia una implementazione corretta della specifica XSLT). Per l'uso delle proprietà definite come argomenti del metodo apply, UWOBO modifica una copia temporanea dell'ultimo foglio di stile della catena, creando o modificando opportunamente l'elemento ⟨xsl:output⟩ ivi contenuto, e usa poi tale copia per eseguire l'ultima trasformazione della catena.

4.5 Problemi di sviluppo

Lo sviluppo di UWOBO non ha presentato particolari problemi implementativi. Molti problemi si sono però presentati in fase di debugging a causa di bug e limitazioni del processore XSLT utilizzato e del parser XML su cui questo si basa.

In particolare sono stati riscontrati due problemi:

- non rientranza del parser XML utilizzato dal processore XSLT
- errata gestione delle URI sottoposte a *URI escaping* un numero di volte superiore a uno

La prima limitazione è ancora oggetto di indagine. Si è comunque riscontrato che l'applicazione di stylesheet XSLT utilizzando il processore libxslt (basato a sua volta sul parser libxml) e l'implementazione DOM libgdome2 non è rientrante. Questa limitazione ha imposto la scelta di una architettura multiprocesso (come descritto in sez. 4.2) piuttosto che di una architettura multithread che avrebbe semplificato notevolmente la gestione degli stylesheet grazie alla primitive di comunicazione tra thread offerte dalla libreria standard OCaml e alla condivisione di memoria tra thread.

La seconda limitazione è stata risolta fornendo una patch all'autore di libxml. La patch è stata integrata in libxml a partire dalla versione 2.5.2. La limitazione consisteva nell'errata gestione di parti di URI (quali gli argomenti passati a UWOBO utilizzando il metodo GET del protocollo HTTP) che erano sottoposte a URI escaping più di una volta. Su queste particolari stringhe la funzione che effettua l'URI escaping non risultava essere l'inversa della funzione che effettua l'URI unescaping. Questo fenomeno dava luogo ad una perdita di informazioni che rendeva impossibile risalire alla risorsa specificata dall'utente.

4.6 Confronto con le precedenti implementazioni

La precedente implementazione di UWOBO fu scritta nel linguaggio di programmazione Java e utilizzava come piattaforma di supporto l'*application server* Tomcat[7]. Il processore XSLT utilizzato dalla precedente implementazione era Xalan[8] basato sul parser XML Xerces[9].

[7] `http://jakarta.apache.org/tomcat/`
[8] `http://xml.apache.org/xalan-j/`
[9] `http://xml.apache.org/xerces-j/`

L'attuale implementazione OCamldi UWOBO si è mostrata superiore sotto molti aspetti, in particolare:

- flessibilità
- prestazioni
- occupazione di memoria
- scalabilità

Per quanto riguarda la **flessibilità** dell'implementazione è sufficiente osservare che l'attuale implementazione di UWOBO è interamente standalone, non richiede alcuna applicazione di supporto quale un web server o simili ed è completamente portabile su qualsiasi architettura per la quale sia disponibile un interprete di bytecode OCaml. La precedente implementazione gode indubbiamente degli stessi vantaggi di portabilità, ma necessita al contrario di un web server installato e dell'application server Tomcat.

Per descrivere le differenze di **prestazioni** utilizzeremo qualche figura di merito: la resa del termine cic:/Coq/Reals/Rlimit/limit_mul.con.body richiedeva con la precedente implementazione 15 secondi affinché il primo output giungesse al browser e 27 secondi per la resa totale. Utilizzando l'attuale implementazione i tempi sono scesi rispettivamente a 11 e 13 secondi. La scarsa differenza nei tempi di generazione del primo output è dovuto al differente approccio dei parser XML sottostanti ai processori XSLT utilizzati: Xerces è un parser ad eventi e genera quindi output incrementale mentre libxml è un parser che lavora su rappresentazioni ad albero dei documenti XML.

L'**occupazione di memoria** della nuova implementazione risulta enormemente inferiore (poche unità di Mb rispetto a circa 100 Mb) a quella della precedente implementazione Java. Ciò è inequivocabilmente imputabile alla necessità della precedente implementazione di "appoggiarsi" a Tomcat (è importante sottolineare che Tomcat non era utilizzato per nessun altro scopo all'interno del progetto HELM se non quello di eseguire UWOBO).

Uno dei riflessi maggiormente negativi dell'esosa occupazione di memoria della precedente implementazione era il suo impatto sulla **scalabilità** di UWOBO. L'implementazione Java di UWOBO infatti non riusciva a portare a termine la resa di molte prove della libreria di oggetti CIC fallendo per esaustione di memoria. L'attuale implementazione OCaml completa con successo la resa degli stessi oggetti a parità di memoria disponibile.

4.7 Implementazione

UWOBO è composto da 5 moduli (fig. 4.2). L'implementazione consta di poco più di 1100 righe di codice OCaml comprensive di commenti.

L'implementazione ha richiesto circa 160 ore/uomo di lavoro. Buona parte del lavoro è stata impiegata nel debugging di libxml/libxslt e nella successiva ristrutturazione da una architettura multi threaded ad una architettura multi processo.

Capitolo 5

drawGraph: generazione di grafi di dipendenza

La disponibilità di una libreria di matematica formalizzata estesa come quella del progetto HELM e la possibilità di esprimere meta informazioni sui documenti in essa contenuta ha permesso di studiare relazioni interessanti tra documenti. Una delle applicazioni più utili derivanti dallo studio di queste relazioni è la possibilità di generare grafi di dipendenza tra termini presenti all'interno della libreria.

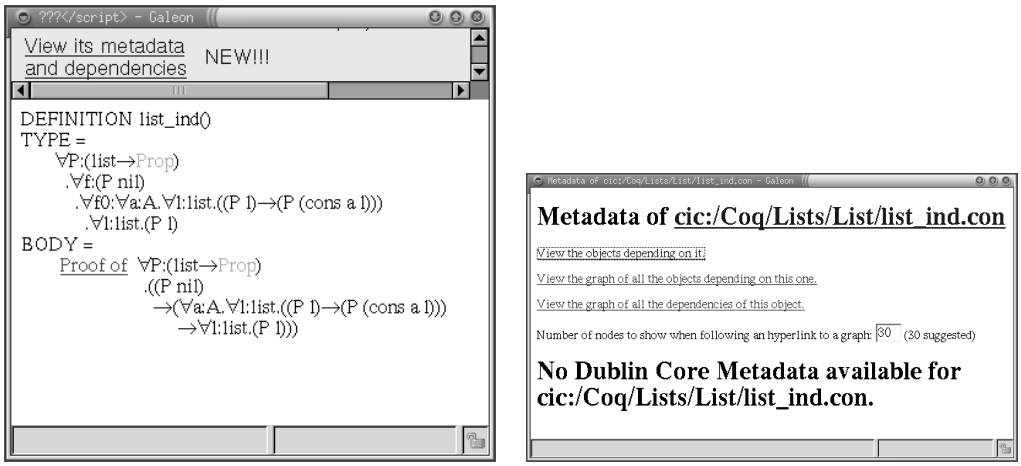
Lo studio di questi grafi di dipendenza può risultare utile ad esempio nel caso in cui si decida di riformalizzare alcune parti della libreria permettendo di sapere quali parti della stessa risulterebbero interessate alla modifica. Analogamente possiamo immaginare di volere estrarre il sottinsieme della libreria riguardante la teoria degli insiemi. Studiando i grafi di dipendenza possiamo risalire a tutti i termini necessari affinché il sottoinsieme creato sia chiuso rispetto alla relazione "dipende da".

Molte applicazioni sono immaginabili a partire da queste relazioni di dipendenza, ma anche la sola visualizzazione degli stessi risulta molto istruttiva. Per questo motivo nell'ambito del progetto HELM è stata realizzata una semplice interfaccia web che permette agli utilizzatori di visualizzare i grafi di dipendenza degli oggetti che stanno visionando.

5.1 Tipologie di grafi

Per la creazione dei grafi di dipendenza abbiamo utilizzato metadati appartenenti a due schemi RDF:

- *helm:rdf:forward*



(a) (b)

Figura 5.1: Principio di induzione su liste e accesso ai relativi metadati

- *helm:rdf:backward*

Il primo associa ad ogni termine presente nella libreria l'insieme delle URI di termini referenziati al suo interno. I metadati di questo tipo danno quindi informazioni quali le prove dalle quali altre prove dipendono. È ad esempio il caso dei lemmi: una dimostrazione che utilizzi uno o più lemmi avrà associati nello schema forward le URI dei termini corrispondenti ai lemmi utilizzati per il suo svolgimento.

Il secondo (backward) rappresenta la relazione inversa: associa cioè ad ogni termine un insieme di URI corrispondenti ai termini che lo referenziano. Nel precedente esempio dei lemmi ad ognuno di essi sarà associata la lista di URI referenzianti prove che li utilizzano.

L'interfaccia web realizzata per la visualizzazione dei grafi supporta due diverse tipologie di grafi accessibili dalla pagina di visualizzazione dei metadati di ogni oggetto. A titolo di esempio utilizzeremo la formalizzazione del principio di induzione su liste[1] riportato in fig. 5.1(a). La rispettiva pagina di visualizzazione dei metadati è riportata in fig. 5.1(b).

Il link "view the graph of all the dependencies of this object" permette di visualizzare il grafo *forward* (fig. 5.2) ovvero di tutti gli oggetti utilizzati dalla dimostrazione del principio di induzione su liste. Il link "view the graph of all objects depending on this one" permette invece di visualizzare il grafo *backward* (5.3) ovvero di tutti gli oggetti che utilizzano il principio di induzione su liste.

[1] *cic:/Coq/Lists/List/list_ind.con*

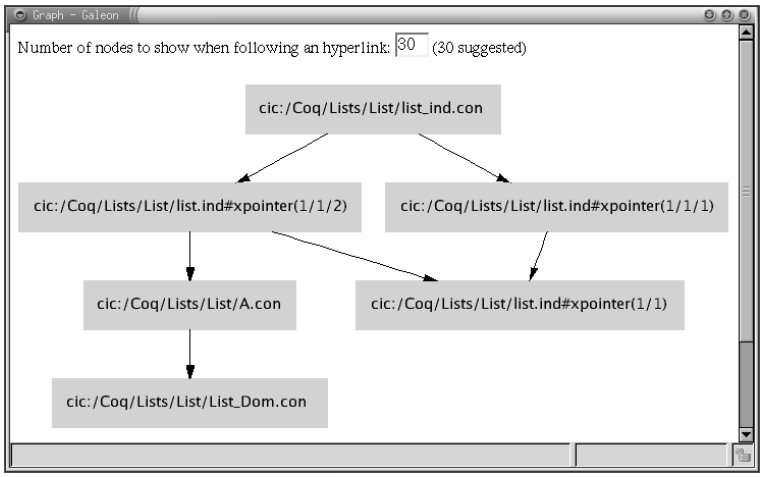


Figura 5.2: Principio di induzione su liste: dipendenze *forward*

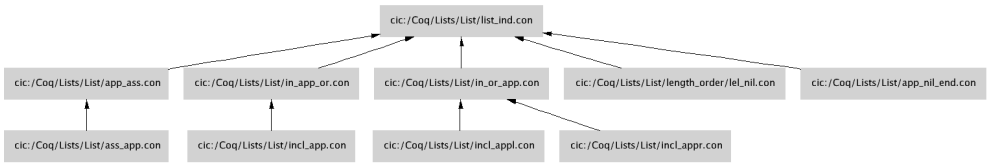


Figura 5.3: Principio di induzione su liste: dipendenze *backward*

5.2 Architettura

L'architettura del generatore di grafi di dipendenza è articolata su quattro servizi web: il getter (cap. 3), UWOBO (cap. 4), *drawGraph* (sez. 5.3) e *uriSetQueue* (sez. 5.4). Il diagramma di interazione tra loro è riportato in fig. 5.4.

L'inizio della sequenza che porta alla generazione di un grafo di dipendenza è una richiesta effettuata dal browser ad UWOBO che applica gli stylesheet che permetteranno la navigazione del grafo. Questi stylesheet (principalmente makeGraphLinks.xsl) aggiungono codice Javascript ai grafi generati che permette di utilizzare menu contestuali su ogni nodo del grafo.

La URI del documento da trasformare è una richiesta di tipo *draw* a drawGraph che restituirà una pagina XHTML contenente un riferimento ad una immagine GIF (il grafo di dipendenza). La URL di tale immagine è una richiesta *get_gif* a drawGraph che referenzia un file all'interno del file system creato durante la precedente invocazione di draw.

L'esecuzione della richiesta draw da parte di drawGraph prevede una ulteriore invocazione ad UWOBO che applica il foglio di stile mk_dep_graph.xsl

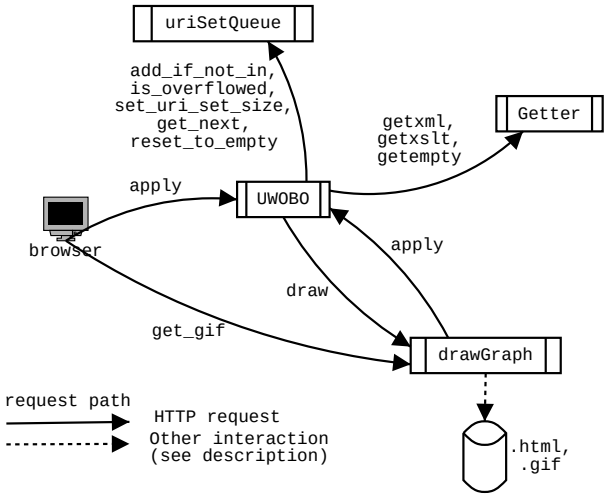


Figura 5.4: Generatore di grafi di dipendenza: architettura

al documento XML "nullo" (restituito dal metodo getempty del getter, sez. 3.4.10). drawGraph passa a questa invocazione di UWOBO tutti i parametri necessari a generare una rappresentazione Graphviz (sez. 5.2.1) del grafo di dipendenza quali. Questi parametri includono, tra gli altri, la URI del documento della libreria origine del grafo ed il tipo di grafo richiesto (forward o backward).

Per la generazione della rappresentazione Graphviz del grafo, UWOBO utilizza un servizio web ausiliario: uriSetQueue. Questo servizio è responsabile di gestire code di URI di lunghezza limitata senza ripetizioni. Le code vengono utilizzate per la visita del grafo globale di dipendenza (costruito seguendo le informazioni contenute nei metadati), il fatto che siano di lunghezza limitata viene utilizzato per limitare la dimensione del grafo restituito all'utente che è solitamente interessato solamente alla parte del grafo "vicina" (in termini di numero di archi) al documento visualizzato.

Una volta ricevuta la rappresentazione Graphviz del grafo, drawGraph genera, utilizzando il tool `dot`, la GIF del grafo corrispondente ed una pagina XHTML. L'immagine GIF viene salvata su disco ed associata ad un identificatore univoco[2]. La pagina XHTML viene restituita come risultato della richiesta draw. La pagina XHTML restituita contiene inoltre riferimenti ad una mappa server side generata utilizzando dot che permette all'utente di utilizzare i nodi del grafo con hyperlink per ottenere la resa di altri grafi di dipendenza.

La pagina XHTML include l'immagine GIF utilizzando l'usuale elemento ⟨IMG⟩ il cui attributo URL corrisponde all'invocazione del metodo get_-

[2]attualmente il PID del processo che sta gestendo la richiesta

gif di drawGraph. L'invocazione get_gif include un parametro che identifica univocamente l'immagine generata in precedenza.

5.2.1 Graphviz

Graphviz[3] ([GKNK93], [GN99]) è un insieme di applicazioni software sviluppate dai laboratori di ricerca di AT&T per la rappresentazione di grafi (sia orientati che non orientati) in diversi formati a partire da una loro descrizione in un semplice linguaggio testuale, sviluppato appositamente.

L'algoritmo di rappresentazione è stato pensato per essere efficiente al punto di essere utilizzabile interattivamente senza alcuna necessità di pregenerare i grafi. La nostra esperienza conferma questa desiderabile qualità.

La rappresentazione testuale dei grafi si base su un formato molto semplice che utilizza una sintassi simile a quella del linguaggio C. A titolo di esempio riportiamo in fig. 5.5 la descrizione testuale del grafo (orientato) di dipendenza dei moduli del broker di HBugs (sez. 7.2.3), la corrispondente rappresentazione grafica è riportata in fig. 7.5.

La rappresentazione testuale inizia con l'asserzione (riga 1) che il grafo descritto è un grafo orientato (il prefisso "di" corrisponde a "direct") denominato "G". La necessità di racchiudere la descrizione del grafo in un blocco delimitato da parentesi graffe deriva dalla possibilità offerta da Graphviz di descrivere più grafi all'interno di un unico file. Le istruzioni successive (righe 2–6) impostano alcuni parametri globali per il grafo quali la dimensione, l'orientamento, ecc.

Le istruzioni che seguono alternano descrizioni di nodi (righe 7, 12, 16, 18–22) a descrizioni di archi (righe 8–11, 13–15, 17, 23). Le descrizioni di nodi riportano il nome del nodo (utilizzato sia per referenziare il nodo nelle descrizioni degli archi sia come etichetta nella rappresentazione del grafo) e sue proprietà quali il tipo di nodo, il colore, ecc. Le descrizioni di archi associano semplicemente due nodi distinguendo quale di essi sia la sorgente e quale la destinazione dell'arco.

La semplicità di questa descrizione testuale rende possibile generarla facilmente in automatico quando necessario. Nel nostro caso la generazione della descrizione testuale dei grafi è delegata al foglio di stile mk_dep_graph.xsl.

[3] `http://www.research.att.com/sw/tools/graphviz/`

```
digraph G {
 size="10,7.5";
 ratio="fill";
 rotate=90;
 fontsize="12pt";
 rankdir = TB ;
 "Hbugs_broker" [style=filled, color=darkturquoise];
 "Hbugs_broker" -> "Hbugs_types";
 "Hbugs_broker" -> "Hbugs_messages";
 "Hbugs_broker" -> "Hbugs_id_generator";
 "Hbugs_broker" -> "Hbugs_broker_registry";
 "Hbugs_broker_registry" [style=filled, color=darkturquoise];
 "Hbugs_broker_registry" -> "ThreadSafe";
 "Hbugs_broker_registry" -> "Hbugs_types";
 "Hbugs_broker_registry" -> "Hbugs_misc";
 "Hbugs_common" [style=filled, color=darkgoldenrod2];
 "Hbugs_common" -> "Hbugs_types";
 "Hbugs_id_generator" [style=filled, color=darkgoldenrod2];
 "Hbugs_misc" [style=filled, color=darkgoldenrod2];
 "Hbugs_types" [style=filled, color=darkgoldenrod2];
 "ThreadSafe" [style=filled, color=darkgoldenrod2];
 "Hbugs_messages" [style=filled, color=darkgoldenrod2];
 "Hbugs_messages" -> "Hbugs_types";
}
```

Figura 5.5: Graphviz: descrizione testuale di un grafo orientato

5.3 Interfaccia di *drawGraph*

5.3.1 Metodo *draw*

Parametri : richiede un unico parametro *url* che referenzia una risorsa contenente la rappresentazione Graphviz del grafo da rappresentare. Tipicamente questo parametro codifica una richiesta ad UWOBO di applicazione del foglio di stile mk_dep_graph.xsl

Richiesta HTTP GET : */draw?url=URL*

Valore di ritorno : una pagina XHTML che referenzia una immagine corrispondente alla rappresentazione del grafo descritto dalla risorsa indicata nel parametro url. L'immagine è referenziata attraverso una richiesta get_gif a questo servizio web

Descrizione : data una descrizione Graphviz di un grafo orientato restituisce

una pagina XHTML che referenzia una rappresentazione GIF del grafo stesso ottenuta utilizzando dot. L'invocazione di questo metodo crea inoltre l'immagine GIF e la salva su disco associandole un identificatore univoco

5.3.2 Metodo *get_gif*

Parametri : richiede un unico parametro *pid* utilizzato per referenziare una immagine GIF precedentemente generata durante l'invocazione del metodo draw

Richiesta HTTP GET : */get_gif?pid=PID*

Valore di ritorno : l'immagine GIF corrispondente all'identificatore univoco specificato dal parametro pid

Descrizione : questo metodo viene tipicamente invocato dal browser nel momento in cui presenta all'utente la pagina XHTML restituita dal metodo draw. Tale pagina contiene infatti un riferimento ad una immagine la cui URL è una invocazione a get_gif. L'esecuzione di questo metodo restituisce al browser l'immagine GIF del grafo creato in precedenza e la rimuove dal file system

5.3.3 Metodo *help*

Parametri : nessuno

Richiesta HTTP GET : */help*

Valore di ritorno : un messaggio testuale contenente informazioni sullo stato di drawGraph e una descrizione delle modalità di utilizzo dei suoi metodi

Descrizione : restituisce un messaggio di utilizzo e altre informazioni sulla corrente esecuzione di drawGraph

5.4 Interfaccia di *uriSetQueue*

5.4.1 Metodo *set_uri_set_size*

Parametri : richiede un parametro *pid* utilizzato per identificare una coda e un parametro numerico *size* indicante la nuova dimensione della coda

Richiesta HTTP GET : */set_uri_set_size?pid=PID&size=SIZE*

Valore di ritorno : un documento XML ben formato costituito da un unico elemento vuoto ⟨done⟩

Descrizione : nel caso in cui la coda specificata da pid non esista crea una nuova coda associata a quell'identificatore e ne imposta la dimensione a size. Nel caso in cui la coda esista reimposta la sua dimensione a size

5.4.2 Metodo *add_if_not_in*

Parametri : richiede un parametro *pid* identificante la coda alla quale si vuole aggiungere una URI ed un parametro *uri* corrispondente alla URI da aggiungere

Richiesta HTTP GET : */add?pid=PID&uri=URI*

Valore di ritorno : può ritornare tre diversi risultati. Ognuno di essi è un documento XML ben formato contenente un unico elemento vuoto. Nel caso in cui la coda pid abbia già raggiunto la sua dimensione massima (i.e. la coda pid è in *overflow*), viene restituito l'elemento ⟨not_added_because_already_too_many⟩; nel caso in cui la coda non sia in overflow ma la URI esista già al suo interno, viene restituito l'elemento ⟨already_in⟩; nel caso in cui la coda non sia in overflow e la URI non esista già al suo interno, viene restituito l'elemento ⟨added⟩

Descrizione : questo metodo è responsabile dell'aggiunta di URI a code precedentemente create. Una URI può essere aggiunta solamente se la coda destinazione non è in overflow e se la URI stessa non è già presente all'interno della coda

5.4.3 Metodo *get_next*

Parametri : richiede un unico parametro *pid* indicante la coda dalla quale si vuole estrarre una URI

Richiesta HTTP GET : */get_next?pid=PID*

Valore di ritorno : nel caso in cui la coda sia vuota, viene restituito un elemento XML vuoto ⟨empty⟩; nel caso in cui la coda non sia vuota e la coda non sia in overflow, viene restituito un elemento XML vuoto ⟨uri⟩ con attributo value contenente la prima URI della coda; nel caso in cui

la coda non sia vuota ma sia in overflow, viene restituito un elemento ⟨marked_uri⟩ analogo al precedente

Descrizione : ritorna la prima[4] URI disponibile all'interno di una coda specificando inoltre se la coda sia in overflow o meno. L'invocazione di questo metodo non causa mai il passaggio di stato da overflow a non-overflow. Ciò implica che una coda che entri nello stato overflow vi rimane fino a quando non viene distrutta

5.4.4 Metodo *is_overflowed*

Parametri : richiede un unico parametro *pid* indicante la coda della quale si vuole conoscere lo stato

Richiesta HTTP GET : */is_overflowed?pid=PID*

Valore di ritorno : restituisce un documento XML formato da un unico elemento vuoto, l'elemento è ⟨true⟩ nel caso in cui la coda pid sia attualmente in overflow, ⟨false⟩ altrimenti

Descrizione : predicato che valuta lo stato di overflow o meno di una coda

5.4.5 Metodo *reset_to_empty*

Parametri : richiede un unico parametro *pid* indicante la coda da distruggere

Richiesta HTTP GET : */reset_to_empty?pid=PID*

Valore di ritorno : un documento XML formato da un unico elemento vuoto ⟨done⟩

Descrizione : distrugge una coda

5.4.6 Metodo *help*

Parametri : nessuno

Richiesta HTTP GET : */help*

Valore di ritorno : un messaggio testuale contenente informazioni sullo stato di uriSetQueue e una descrizione delle modalità di utilizzo dei suoi metodi

[4] le code sono gestite secondo politica FIFO

Descrizione : restituisce un messaggio di utilizzo e altre informazioni sulla corrente esecuzione di uriSetQueue

5.5 Implementazione

Sia drawGraph che uriSetQueue sono stati implementati in un singolo file sorgente. Complessivamente la loro implementazione consta di 310 righe di codice OCaml.

L'implementazione ha richiesto circa due giorni lavorativi (16 ore/uomo).

Capitolo 6

searchEngine: ricerche all'interno della libreria

Una delle maggiori potenzialità offerte dalla strutturazione in XML dei documenti che compongono la libreria del progetto HELM è la possibilità di effettuare ricerche all'interno di essa.

A questo scopo, nell'ambito del progetto, è stato sviluppato il linguaggio MathQL ([GS03], [Gui03]) utilizzando il quale è possibile descrivere query per la ricerca di oggetti all'interno della libreria. È ad esempio possibile:

- cercare tutti gli oggetti che utilizzano il principio di induzione sui naturali nel corpo della dimostrazione
- cercare tutti gli oggetti che dimostrino proprietà commutative
- cercare tutti gli oggetti che utilizzano come ipotesi il fatto che la funzione

  $$f(x) = \sqrt{x^2 + 1}$$

  definita su R sia monotona crescente.

Il motore in grado di interpretare MathQL e di utilizzare il database dei metadati per eseguire effettivamente le query e generarne i risultati è implementato in OCaml.

Questo motore è stato fino ad ora utilizzato per l'implementazione di funzionalità del proof assistant; utilizzando gTopLevel è infatti possibile effettuare query e visualizzarne i risultati. Fino ad ora non era però disponibile nessun'altra modalità di accesso a queste funzionalità di ricerca.

Abbiamo quindi pensato di implementare un servizio web che ponga requisiti di usabilità minimi e che permetta l'utilizzo delle funzionalità di ricerca. Questo servizio web è stato chiamato *searchEngine* ed è utilizzabile, congiuntamente ad una interfaccia in corso di realizzazione ([Ned03]) in HTML e Javascript, con un semplice browser.

Attualmente è disponibile un metodo di *basso livello* che permette di formulare query MathQL. Questo metodo prende il nome di *execute*. L'utilizzo di questo metodo è consigliato solamente per i conoscitori di MathQL. L'interfaccia web è comunque in grado di aiutare l'utente nel processo di creazione delle query "seguendolo" lungo le produzioni della grammatica di MathQL.

Sono inoltre disponibili due query di più *alto livello*.

La prima, *locate*, è la funzione più semplice di ricerca all'interno della libreria. Richiede come argomento un identificatore testuale e, utilizzando i metadati dello schema helm:rdf:object_name, restituisce una lista di URI. Ogni URI identifica un oggetto all'interno della libreria di HELM che corrisponde all'identificatore fornito.

Cercando ad esempio l'identificatore "list" utilizzando questa query vengono restituite sei URI tra le quali possiamo notare la definizione induttiva di liste (cic:/Coq/Lists/List/list.ind#1/1), la definizione induttiva di liste polimorfe (cic:/Coq/Lists/PolyList/list.ind#1/1) e altre definizioni di liste formalizzate per la dimostrazione di particolari teoremi.

La seconda query di alto livello prende il nome di *searchPattern*. Utilizzando searchPattern è ad esempio possibile cercare all'interno della libreria un teorema (falso!) che affermi che ogni proposizione su liste sia vera:

$$\forall P : (\text{list} \rightarrow \text{Prop})\ \forall L : \text{list}\ (P\ L)$$

oppure, nel linguaggio testuale utilizzato da gTopLevel:

```
!P:list -> Prop.!L:list.(P L)
```

Questa query non produrrà alcun risultato dato che nessun teorema può dimostrare quanto richiesto.

È inoltre possibile cercare *pattern*, ovvero termini non completamente specificati che contengano "buchi". Utilizzando questa tecnica è ad esempio possibile cercare tutti i teoremi contenuti nella libreria che dimostrino che un naturale n è uguale al prodotto di un certo naturale (non specificato) per n:

$$\forall n : \text{nat}\ n = ? * n$$

nel linguaggio testuale di gTopLevel:

```
!n:nat.(eq ?  n (mult ?  n))
```

dove il carattere ? rappresenta un buco[1]. Il risultato di questa query riporta la URI di un unico teorema che dimostra

$$\forall n : \text{nat}\ n = 1 * n$$

6.1 Architettura

Essendo il linguaggio MathQL già stato implementato in moduli OCaml riutilizzabili, l'architettura di searchEngine è risultata triviale. È stato infatti implementato un unico sorgente *searchEngine.ml* che utilizza i moduli esposti dall'implementazione di MathQL per il parsing del linguaggio, il pretty printing dei risultati e l'esecuzione delle query.

Per l'implementazione del binding HTTP del servizio web è stato utilizzato OCamlHTTP analogamente a quanto visto per le altre componenti implementate.

6.1.1 Disambiguazione

L'utilizzo del motore delle funzionalità di ricerca offerte da MathQL è risultato inizialmente molto scomodo per l'utente finale. Questa scomodità risiedeva principalmente nella necessità di specificare la URI di ogni oggetto che occorreva all'interno della query.

Per risolvere queste problematiche è stato implementato dal dott. Claudio Sacerdoti Coen un meccanismo di *disambiguazione* basato sull'utilizzo combinato dei metadati appartenenti allo schema object_name e di type checking.

Non ci soffermeremo sui dettagli della disambiguazione. Cercheremo solamente di dare una idea generale del suo funzionamento.

A titolo di esempio consideriamo il termine (riportato nel linguaggio testuale utilizzato da gTopLevel):

```
(eq ? (plus (mult n m) n))
```

Questo termine risulta *ambiguo* in quanto riporta un buco (il carattere "?") e due identificatori (plus e mult) per ognuno dei quali non è noto il corrispondente oggetto all'interno della libreria.

[1] la rappresentazione testuale risulta in questo caso più precisa e riporta due buchi: il primo rappresenta l'insieme di definizione dell'uguaglianza, il secondo il naturale non specificato

Il processo di disambiguazione, dato un termine di questo tipo, deve essere in grado di istanziare i buchi e di associare ad ogni identificatore la URI di un oggetto della libreria.

L'idea sottostante al processo di disambiguazione è di associare ad ogni identificatore tutti i possibili oggetti corrispondenti utilizzando i metadati object_name e di scartare utilizzando il type checking le combinazioni non valide. Notiamo infatti che non tutte le combinazioni sono possibili: supponendo ad esempio che a plus corrisponda sia una somma sui naturali che una somma sui reali e che lo stesso valga per mult; non possiamo associare al primo la somma sui naturali ed al secondo la somma sui reali, il termine risultante non risulterebbe correttamente tipato.

L'implementazione triviale di questa idea richiederebbe un tempo computazionale proporzionale al numero di possibili combinazioni di URI e risulterebbe troppo dispendioso in termini di tempo. L'implementazione attuale richiede invece tempo computazionale quasi lineare, nel caso medio, nel numero di oggetti ambigui presenti in un termine. Questo risultato è stato ottenuto procedendo alla analisi delle possibili istanziazioni di un oggetto per volta scartando così "in anticipo" le istanziazioni non tipabili degli altri oggetti.

Se ad esempio l'unica istanziazione possibile di plus fosse la somma sui naturali, n deve necessariamente essere un numero naturale e risulta inutile valutare tutte le istanziazioni di mult diverse dalla moltiplicazione sui naturali.

Dal punto di vista dell'utilizzo, il processo di disambiguazione è composto da due fasi:

1. selezione delle possibili istanziazioni di ogni identificatore

2. selezione delle possibili interpretazioni

Nella prima fase viene richiesto all'utente di selezionare quali, tra le possibili istanziazioni di ogni identificatore, vuole considerare durante il processo di disambiguazione. L'utente può in questa fase selezionare uno o più URI per ogni identificatore.

Nella seconda fase, vengono presentate all'utente tutte le possibili interpretazioni del termine ambiguo fornito. È infatti possibile che esistano più interpretazioni possibili di un termine ambiguo.

La query searchPattern di searchEngine permette all'utente l'inserimento di termini ambigui che saranno poi sottoposti al processo di disambiguazione. Praticamente questo processo è implementato come *dialogo* tra l'interfaccia web e searchEngine. Il dialogo continua fino a quando searchEngine non riceve, come

argomenti addizionali del metodo searchPattern, tutte le informazioni necessarie a disambiguare tutti i termini che compaiono nella query.

Nel caso in cui queste informazioni non siano disponibili searchEngine ritorna all'utente un form HTML che richiede parte delle informazioni mancanti (può ad esempio presentare una lista di URI per progredire nella fase 1 o una lista di interpretazioni per progredire nella fase 2). L'interfaccia web si preoccupa di tenere traccia delle informazioni passate e di spedirle volta per volta a searchEngine.

6.2 Interfaccia

L'interfaccia di searchEngine prevede tre metodi corrispondenti a tre diverse tipologie di query (*execute, locate, searchPattern*). È inoltre presente un metodo aggiuntivo (*getpage*) utilizzato per restituire all'utente le pagine che compongono l'interfaccia Web alle funzionalità del motore di ricerca[2].

6.2.1 Metodo *execute*

Parametri : richiede un unico parametro *query* corrispondente ad una rappresentazione testuale di una query MathQL. Il formato di tale rappresentazione è quello richiesto dalla funzione *query_of_text* del modulo MQueryGenerator

Richiesta HTTP GET : */execute?query=QUERY*

Valore di ritorno : una pagina HTML contenente il risultato testuale dell'esecuzione della query. Attualmente il risultato è restituito in un formato non strutturato incluso all'interno dell'elemento HTML ⟨PRE⟩. Un formato più strutturato è attualmente in fase di studio.

Descrizione : è il metodo più a basso livello tra quelli che permettono di effettuare query. È infatti necessario conoscere il linguaggio testuale MathQL per fornire un argomento query opportuno al metodo. Questo metodo esegue la query ricevuta e restituisce una rappresentazione testuale dei risultati ottenuti all'interno di una pagina HTML

[2]getpage è stato implementato per superare alcune limitazioni di Javascript. Risulta però molto utile anche nel caso in cui non si disponga di un server HTTP utilizzabile per la gestione delle pagine HTML che compongono l'interfaccia web al motore di ricerca

6.2.2 Metodo *locate*

Parametri : richiede un unico argomento *id* rappresentante un identificatore testuale del quale si vogliono cercare occorrenze nella libreria di HELM

Richiesta HTTP GET : */locate?id=ID*

Valore di ritorno : restituisce una rappresentazione testuale di tutte le occorrenze dell'identificatore id riscontrate nella libreria. Tale rappresentazione testuale viene restituita all'interno di una pagina HTML

Descrizione : questo metodo implementa la query di alto livello *locate*. locate utilizza lo schema di metadati helm:rdf:object_name per associare ad un identificatore testuale le URI di tutti gli oggetti contenuti nella libreria che possono corrispondervi. I risultati vengono restituiti all'utente all'interno di una pagina HTML

6.2.3 Metodo *searchPattern*

Parametri : richiede un parametro obbligatorio *term* che contiene una rappresentazione testuale del termine da cercare. Il termine può contenere buchi ed essere ambiguo. I parametri opzionali *aliases* e *choices* vengono utilizzati durante il processo di disambiguazione. aliases contiene le associazioni tra identificatori e URI già disambiguate: ad ogni identificatore viene associata la sola URI dell'unico termine corrispondente possibile. choices contiene le possibili istanziazioni degli identificatori non ancora disambiguati: ad ogni identificatore viene associata una lista di URI di possibili termini

Richiesta HTTP GET :

/searchPattern?term=TERM
[&aliases=alias%20KEY%20VALUE[alias%20KEY%20VALUE[...]]]
[&choices=CHOICE1[%20CHOICE2[...]][;CHOICE1[%20...]]]

Valore di ritorno : nel caso in cui il termine specificato da term non sia ambiguo ritorna una pagina HTML contenente una lista di URI. Ad ogni URI corrisponde un termine presente all'interno della libreria di HELM che soddisfa il pattern specificato. Nel caso in cui il termine sia ambiguo vengono ritornate pagine HTML che, presentate all'utente in uno dei frame dell'interfaccia web al motore di ricerca, permettono di progredire nel processo di disambiguazione

Descrizione : implementa la query di alto livello searchPattern e l'eventuale processo di disambiguazione necessario. Dato un pattern su termini restituisce le URI degli oggetti della libreria che soddisfano il pattern. Le URI sono restituite all'interno di una pagina HTML

6.2.4 Metodo *getPage*

Parametri : richiede un unico argomento *url* che identifica la pagina richiesta

Richiesta HTTP GET : */getpage?url=URL*

Valore di ritorno : la pagina HTML identificata da url

Descrizione : questo metodo fornisce su richiesta le pagine HTML che compongono l'interfaccia web al motore di ricerca. Le pagine sono identificate dal parametro url. Questo metodo permette di utilizzare il motore di ricerca con un semplice browser anche nel caso in cui non sia disponibile un server HTTP. Permette inoltre di superare alcune limitazioni di Javascript[3], linguaggio nel quale è implementata la parte "attiva" dell'interfaccia

6.3 Implementazione

Il motore di ricerca è stato realizzato in unico file sorgente che consta di circa 330 riga di codice OCaml.

La sua implementazione ha richiesto circa 30 ore/uomo ed è stata realizzata collaborando con gli implementatori dell'interfaccia web al motore di ricerca e di MathQL.

[3]non è possibile comunicazione Javascript tra frame che provengano da server diversi

Capitolo 7

HBugs: supporto alla dimostrazione interattiva

L'approccio alla dimostrazione dei proof assistant prevede la ripetizione di una sequenza di passi che portino alla dimostrazione di un determinato *goal*: in particolare all'utente viene richiesto di specificare il goal (l'obiettivo della dimostrazione) e di indicare al proof assistant la sequenza di tattiche (una per volta) necessarie alla dimostrazione dello stesso. La sequenza di tattiche necessarie a concludere una prova prende il nome di *script*.

L'aiuto che il proof assistant offre all'utente è quello di mostrare, dopo l'applicazione di ogni tattica, lo stato attuale della prova, ovvero quanto di essa sia già stato dimostrato e quanto ancora resti da dimostrare. Ognuno di questi passaggi è controllato mediante proof checking garantendo così all'utente l'impossibilità di generare passaggi errati nella dimostrazione.

Risulta evidente che, sebbene i vantaggi dei proof assistant siano notevoli, il gravoso compito della scelta delle tattiche resta interamente a carico dell'utente. Il processo di dimostrazione non è infatti automatizzabile in quanto indecidibile nel caso generale.

Questo scenario è inoltre peggiorato dalla presenza di tattiche molto potenti, la cui applicazione però risulta particolarmente pesante e richiede molto tempo prima che il proof assistant sia in grado di restituire il successivo stato della prova o anche solamente un messaggio di errore inerente alla non applicabilità della tattica scelta dall'utente. In questa tipologia di tattiche risultano presenti, ad esempio, tattiche riflessive quali `Ring` e `Fourier` del proof assistant Coq ([BBC+97]).

A titolo di esempio riportiamo che l'applicazione della tattica Ring all'interno del proof assistant del progetto HELM richiede diversi minuti per il suo

type checking di tutti i termini da cui la sua applicazione dipende. Questa attesa risulta particolarmente fastidiosa nel caso in cui l'applicazione della tattica fallisca, in quanto l'utente è impossibilitato a procedere nella dimostrazione in attesa della fine del type checking per poi ritrovarsi in uno stato identico al precedente.

Una possibile soluzione a questa categoria di problemi consiste nell'implementazione di una architettura di *suggerimento* per gTopLevel. Questa architettura deve permettere ad uno o più *suggeritori* di lavorare in parallelo all'utente. Ogni qual volta uno di essi riscontra la possibilità di progredire con successo nella dimostrazione applicando una data tattica, lo segnala all'utente che può decidere di seguire o meno il "consiglio" ricevuto.

Una idea simile è stata precedentemente implementata nell'ambito del proof assistant Omega ([BC$^+$]) sotto il nome di Omega Ants ([BS98]). L'approccio di Omega Ants non è però fedelmente riportabile al proof assistant del progetto HELM data la fondamentale differenza nel concetto di *tattica*. All'interno di Omega una tattica è un passo di dimostrazione molto semplice (una o più applicazioni di semplici regole di deduzione naturale). Nel progetto HELM invece il concetto di tattica è mutuato dal proof assistant Coq e può includere azioni molto più complesse quali l'utilizzo di tatticali, di sistemi di riscrittura esterni o di riflessività.

In questo capitolo descriveremo HBugs, il *suggeritore* implementato per il proof assistant del progetto HELM.

7.1 Terminologia

Riportiamo di seguito l'interpretazione dei termini utilizzati per la descrizione di HBugs.

gTopLevel: il proof assistant del progetto HELM

tutor: un processo software responsabile della applicazione di una o più tattiche ad uno stato di gTopLevel in grado di generare e inviare suggerimenti nel caso in cui le tattiche di cui è responsabile siano applicabili a tale stato

broker: un processo software responsabile della mediazione tra gTopLevel e i tutor

hint: un *suggerimento* interpretabile da gTopLevel. Un suggerimento corrisponde sempre ad una possibile interazione dell'utente con gTopLevel. In

generale un suggerimento può indicare l'applicazione di una tattica e gli eventuali argomenti necessari alla applicazione dalla stessa

hint list: la lista di suggerimenti applicabili allo stato attuale di gTopLevel

stato: lo stato di gTopLevel rappresentante la dimostrazione incompleta in corso

musing: la computazione eseguita da un tutor per verificare la possibilità di generare un suggerimento per l'utente. Questa computazione può concludersi con un successo o con un fallimento. Un suggerimento viene inviato all'utente solo in caso di successo

7.2 Architettura

L'architettura di HBugs può essere schematizzata come in fig. 7.1[1].

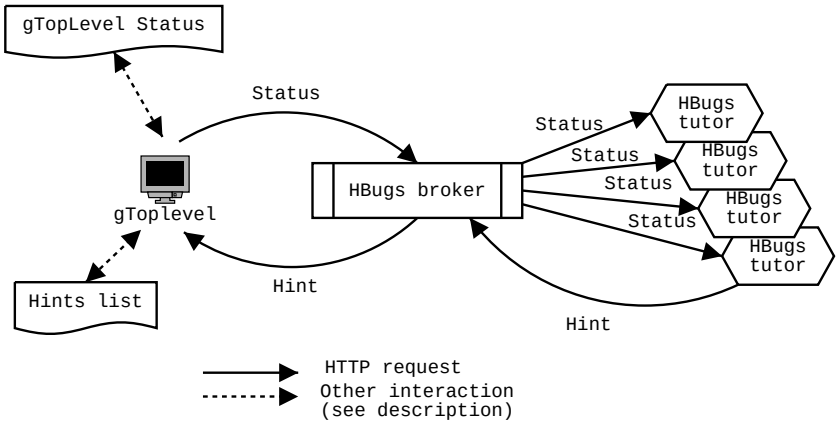


Figura 7.1: HBugs: architettura

Gli attori fondamentali di HBugs sono tre:

- broker
- gTopLevel
- tutor

[1]la figura in realtà mostra le interazioni tra gli attori di HBugs a regime ovvero al termine delle fasi iniziali di registrazione, per una descrizione di queste fasi si veda 7.2.1

Lo scopo fondamentale del **broker** è quello di disaccoppiare gTopLevel dai singoli tutor in modo da fornire a gTopLevel un unico punto di accesso ad HBugs.

gTopLevel è il proof assistant di HELM opportunamente modificato per potere interagire con HBugs. In particolare è stato necessario modificarlo per potere notificare all'utente la ricezione di nuovi suggerimenti e per notificare HBugs ogni qual volta lo stato della dimostrazione cambia.

I **tutor** sono gli agenti software responsabili di elaborare lo stato attuale della dimostrazione e generare, quando possibile, i suggerimenti per l'utente.

7.2.1 Una sessione di HBugs

Per semplicità schematizziamo lo svolgimento di una sessione di utilizzo di HBugs.

Per potere utilizzare HBugs è innanzitutto necessario che sia in esecuzione un broker e che il suo punto di accesso[2] sia noto all'utilizzatore di gTopLevel e ai tutor.

È ora possibile eseguire uno o più tutor configurandoli opportunamente affinché siano in grado di rintracciare il broker. Tipicamente viene eseguito un tutor per ogni tattica della quale si vuole supportare il suggerimento automatico per l'utente. Ciò non è però strettamente necessario, è anche possibile implementare tutor che supportino il suggerimento di più di una tattica.

Il primo compito svolto da un tutor appena posto in esecuzione è la *registrazione* presso il broker. Questa fase è necessaria affinché il broker possa mantenere una lista dei tutor attualmente disponibili e possa segnalarli a gTopLevel su richiesta. All'interno di questa fase inoltre tutor e broker si scambiano i rispettivi identificatori univoci con lo scopo di autenticare i messaggi successivi.

Una volta terminata la fase di registrazione i tutor restano in attesa di richieste di elaborazione da parte del broker.

A questo punto è possibile utilizzare gTopLevel e abilitare, utilizzando la sua interfaccia grafica, il supporto per HBugs. Analogamente a quanto visto per i tutor, la prima fase di utilizzo di gToplevel con supporto per HBugs è la fase di *registrazione*.

Durante la registrazione gTopLevel contatta il broker e avviene lo scambio degli identificatori univoci dei due allo scopo di autenticare i successivi messaggi.

Durante la registrazione gTopLevel contatta il broker inviandogli il suo identificatore univoco; questi risponde con un messaggio analogo. Gli identificato-

[2] una URL che ne identifichi il servizio web

ri così scambiati verranno utilizzati successivamente per l'autenticazione dei messaggi.

La fase successiva prende il nome di *sottoscrizione*. In questa fase gTopLevel richiede al broker la lista dei tutor attualmente disponibili (ovvero i tutor che risultano attualmente registrati presso di lui) e mostra all'utente tale lista. L'utente sceglie da questa lista i tutor da utilizzare e gTopLevel si preoccupa di memorizzare questa informazione ricevuta all'interno di un apposito messaggio di sottoscrizione.

L'utente di gTopLevel può a questo punto procedere normalmente alla dimostrazione di un goal. Ogni qual volta lo stato di gTopLevel cambia (ad esempio quando l'utente effettua un passo di prova utilizzando una tattica), gTopLevel invia un messaggio al broker contenente il nuovo stato e svuota la lista dei suggerimenti attualmente utilizzabili.

Il broker si preoccupa a questo punto di far terminare, inviando un apposito messaggio ai tutor interessati, tutte le *musing* relative al precedente stato di gTopLevel e di attivare nuove musing sui tutor ai quali gTopLevel è attualmente sottoscritto. Questa fase si ripete ogni qual volta lo stato di gTopLevel cambia.

Nel caso in cui una musing termini con successo, il tutor di competenza segnala al broker l'esito e invia a questi una hint per gTopLevel. Il broker si preoccuperà poi di inoltrare il suggerimento a gTopLevel che lo aggiungerà alla lista dei suggerimenti attualmente utilizzabili e notificherà l'utente dell'arrivo di un nuovo suggerimento. L'interfaccia grafica di gTopLevel permette inoltre all'utente di utilizzare facilmente quel suggerimento senza doversi preoccupare dei dettagli del suo contenuto.

Nel caso in cui invece una musing termini con un fallimento il broker ne prende atto e non invia alcun messaggio aggiuntivo a gTopLevel. La lista dei suggerimenti utilizzabile in questo caso rimane immutata[3].

È inoltre possibile da gTopLevel abilitare e disabilitare il supporto per HBugs in modo che le modifiche di stato non vengano segnalate al broker.

7.2.2 HBugs common

HBugs è strutturato in molti moduli raggruppati tra loro in diverse librerie.

La libreria `Hbugs_common` contiene alcune funzionalità comuni utilizzate dagli altri componenti di HBugs.

[3]stiamo studiando la possibilità di raffinare i valori di ritorno delle musing. È infatti possibile che il fallimento di una tattica implichi la non risolvibilità di un goal. In questo caso è auspicabile segnalare all'utente che il goal attuale non è dimostrabile

Descrizione dei moduli

Hbugs_common è composta da cinque moduli, il grafo di dipendenza dei quali è riportato in fig. 7.2.

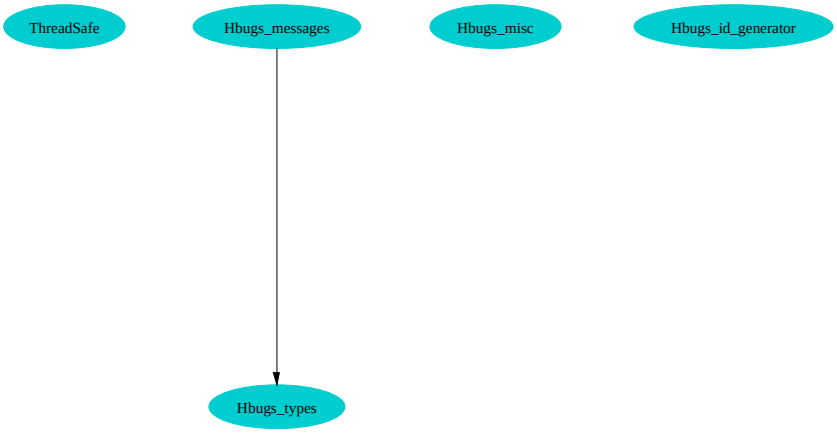


Figura 7.2: Hbugs_common: dipendenza dei moduli

ThreadSafe contiene l'implementazione di una classe (`threadSafe`) che fornisce alcuni metodi utilizzabili in classi da essa derivate per garantire la *thread safety* di uno o più metodi. In particolare è possibile definire metodi che siano protetti da mutua esclusione e metodi che vengano utilizzati secondo il modello di programmazione concorrente *lettori/scrittori*

Hbugs_types definisce i tipi utilizzati per l'implementazione di HBugs. In particolare in questo modulo viene definito il tipo `message` utilizzato per rappresentare tutti i messaggi scambiati tra gli attori di HBugs (si veda sez. 7.3). Contiene inoltre la definizione dei tipi associati allo stato di gTopLevel, ai suggerimenti e agli identificatori univoci dei vari attori

Hbugs_messages implementa le funzioni utilizzate per la serializzazione e la deserializzazione dei messaggi di HBugs e funzioni che implementano l'invio e la ricezione di messaggi tra i vari attori

Hbugs_misc contiene l'implementazione di funzioni non strettamente correlate ad HBugs, ma utilizzate per la sua implementazione. In particolare troviamo all'interno di questo modulo le funzioni che implementano i metodi HTTP GET e POST

Hbugs_id_generator troviamo in questo modulo l'implementazione delle funzioni che generano gli identificatori univoci utilizzati per l'autenticazione dei vari attori di HBugs

7.2.3 HBugs broker

Il **broker** è il processo più importante dell'architettura di HBugs in quanto svolge il ruolo di mediatore tra gTopLevel e i vari tutor disponibili.

Il broker può quindi essere analizzato dai suoi due punti di vista: il rapporto con gTopLevel e il rapporto con i tutor. Sotto entrambi i punti di vista il broker si presenta sia come un servizio web che attende richieste (passivo) sia come un agente software (attivo) che contatta gTopLevel e tutor per inoltrare messaggi ricevuti rispettivamente da tutor e gTopLevel (fig. 7.3).

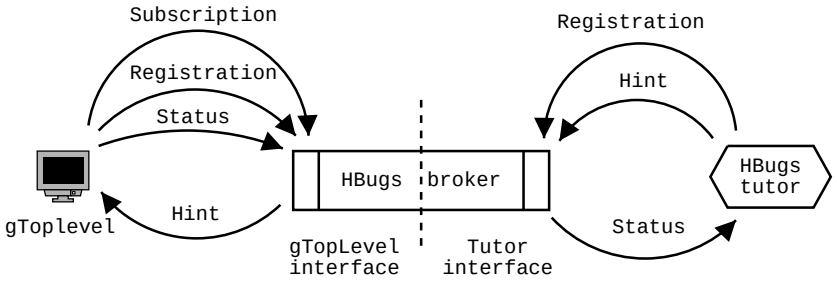


Figura 7.3: Hbugs_broker: interfacce

Dal punto di vista implementativo invece, il broker è composto da un server HTTP utilizzato per la ricezione delle richieste da parte di gTopLevel e tutor, un client HTTP utilizzato per l'inoltro dei messaggi tra gTopLevel e tutor, un insieme di registri utilizzati per memorizzare informazioni quali la lista dei client e dei tutor attualmente registrati e da un motore di serializzazione e deserializzazione dei messaggi di HBugs(fig. 7.4).

Il broker dispone di tre diversi registri:

clients registry utilizzato per memorizzare l'insieme dei client attualmente registrati e la lista di sottoscrizioni di ciascuno di essi

tutors registry utilizzato per memorizzare l'insieme dei tutor attualmente disponibili e altre informazioni ad essi correlate quali il tipo di suggerimenti che possono fornire

musings registry utilizzato per memorizzare la lista delle musing attualmente in esecuzione presso i tutor disponibili

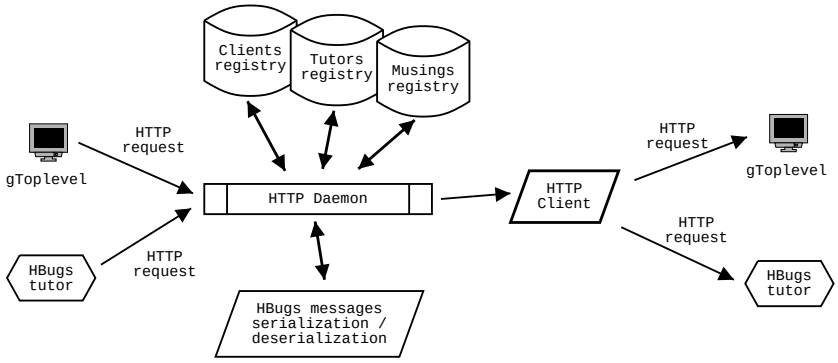


Figura 7.4: Hbugs_broker: architettura

L'**interfaccia presentata a gTopLevel** permette l'esecuzione di tre azioni: registrazione, sottoscrizione e notifica del cambiamento di stato.

La registrazione (**Registration**) si svolge all'atto della prima attivazione del supporto per HBugs da gTopLevel. In questa fase gTopLevel invia un messaggio di registrazione al broker indicando il suo identificatore univoco e l'URL al quale contattarlo. Il broker memorizza queste informazioni nel clients registry e risponde al client comunicandogli il proprio identificatore univoco. Da questo momento in poi ogni scambio di messaggio tra gTopLevel e broker è autenticato, viene cioè controllato che ogni messaggio contenga l'identificatore univoco del mittente precendentemente scambiato.

La fase di sottoscrizione (**Subscription**) prevede che gTopLevel richieda al broker la lista dei tutor attualmente disponibili (ovvero dei tutor che si sono precedentemente registrati presso il broker) e, una volta ricevuta tale lista, invii al broker un messaggio di sottoscrizione indicante quali tutor l'utente di gTopLevel è interessato ad utilizzare ogni qual volta lo stato del proof assistant cambia. Questa fase può essere ripetuta anche più volte nel caso in cui l'utente voglia cambiare l'insieme di tutor utilizzati.

La fase di notifica del cambiamente di stato (**Status**) si ripete ogni qual volta lo stato di gTopLevel cambia. In particolare lo stato cambia al verificarsi di una di queste tre condizione: inizio di una nuova dimostrazione, caricamento di una dimostrazione incompleta salvata in precedenza, esecuzione di un passo di dimostrazione utilizzando una tattica. Durante questa fase gTopLevel invia un messaggio di cambiamento di stato al broker contenente una serializzazione del nuovo stato di gTopLevel. Il broker consulta il musings registry e interrompe tutte le musing attive sullo stato precedente. Infine attiva tutti i tutor ai quali gTopLevel è sottoscritto comunicando loro il nuovo stato.

L'**interfaccia presentata ai tutor** permette l'esecuzione di due azioni distinte: registrazione e notifica dell'esito di una musing.

La registrazione (**Registration**) si svolge per ogni tutor appena questi viene eseguito. In questa fase il tutor invia al broker un messaggio di richiesta di registrazione includendo il suo identificatore univoco ed altre informazioni utili a descriverlo quali: l'URL al quale contattarlo, una descrizione testuale delle sue funzionalità e la formalizzazione del tipo di suggerimento che può fornire a gTopLevel. Il broker registra il tutor aggiungendolo al tutors registry e risponde al tutor comunicando il suo identificatore univoco. Da questo momento in poi ogni messaggio scambiato tra il broker e il tutor appena registrato viene autenticato controllando la correttezza dell'identificatore univoco del mittente.

La notifica dell'esito di una musing (**Hint**) si verifica ogni qual volta un tutor completa una musing, ovvero una sua elaborazione inerente ad un particolare stato della prova precedentemente comunicatogli dal broker. L'esito della musing viene comunicato mediante un messaggio al broker. Nel caso in cui l'esito sia negativo il broker non compie alcuna azione aggiuntiva se non rispondere al tutor per concludere la comunicazione. Nel caso in cui l'esito sia positivo il broker consulta il tutors registry ed invia al client il suggerimento associato al tutor che ha concluso la musing allegando eventualmente parametri opzionali anch'essi contenuti nel messaggio di notifica dell'esito della musing.

Come risulta evidente dalle precedenti descrizioni le comunicazioni "attive" del broker nei confronti di gTopLevel riguardano l'invio di suggerimenti originati a seguito dell'esito positivo di una musing. Analogamente, le comunicazioni attive nei confronti dei tutor consistono nell'invio di messaggi di cambiamento di stato originati da un cambiamento dello stato interno di gTopLevel.

Descrizione dei moduli

Il broker di HBugs è implementato in due moduli:

Hbugs_broker che implementa il server HTTP preposto a gestire le richieste di tutor e gTopLevel

Hbugs_broker_registry che implementa i registri precedentemente descritti in tre classi distinte: `clients`, `tutors` e `musings`

I moduli del broker dipendono inoltre dai moduli di Hbugs_common (sez. 7.2.2). In fig. 7.5 è riportato il grafo di dipendenza dei moduli del broker comprensivo delle dipendenze dai moduli di Hbugs_common.

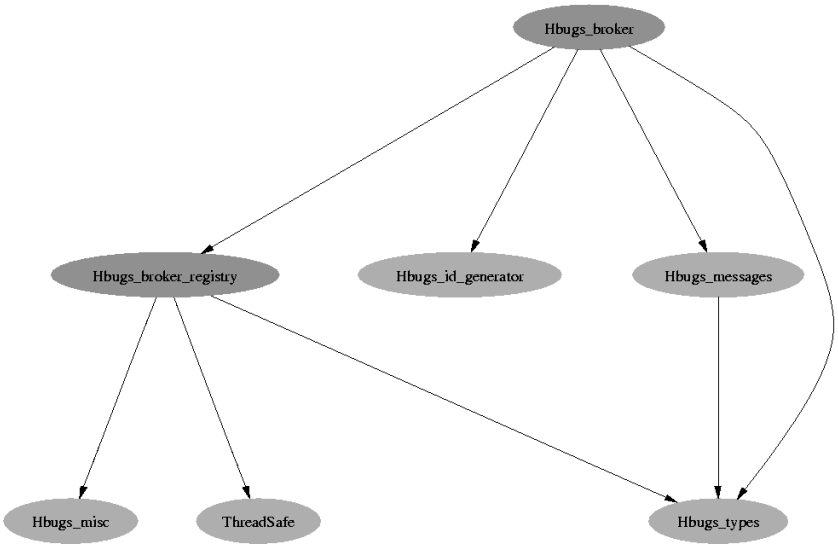


Figura 7.5: Hbugs_broker: dipendenza dei moduli

7.2.4 HBugs tutor

Lo scopo progettuale di HBugs è quello di definire una architettura che permetta l'implementazione di un meccanismo di suggerimenti per gTopLevel. La definizione di singoli tutor esula quindi da tale scopo. In questa sezione definiremo quindi l'architettura di un generico tutor compatibile con HBugs. A titolo di esempio l'implementazione di HBugs realizzata include alcuni tutor utilizzabili come esempi per svilupparne altri.

Ogni singolo tutor utilizzabile da HBugs presenta una interfaccia come servizio web utilizzabile dal broker e utilizza a sua volta i servizi messi a disposizione dall'interfaccia del broker verso i tutor (si veda sez. 7.2.3). L'architettura di un tutor generico è schematizzata in fig. 7.6.

L'interfaccia presentata verso il broker presenta due metodi: uno utilizzato per iniziare una nuova musing e uno utilizzato per terminare una musing attualmente in esecuzione.

Il primo metodo (**inizio di una nuova musing**) viene attivato dal broker inviando al tutor un apposito messaggio contenente una serializzazione dello stato del proof assistant sul quale eseguire la musing. Alla ricezione di questo messaggio il tutor deserializza lo stato ricevuto e prova a progredire nella dimostrazione applicando una tattica a quello stato[4]. L'applicazione della tattica

[4]nulla vieta però di immaginare tutor che provino di progredire nella prova utilizzando più di una tattica

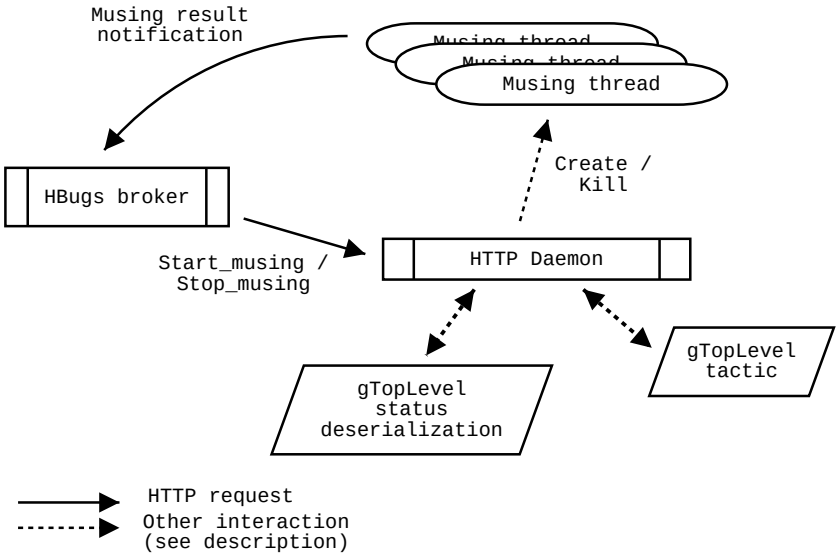


Figura 7.6: Hbugs_tutor: architettura

può richiedere o meno parametri addizionali. L'istanziazione di questi parametri è lasciata all'implementatore del tutor.

L'applicazione della tattica è delegata ad un thread creato per l'occasione e ad ogni musing viene associato un identificatore univoco. Dopo avere creato il thread responsabile della musing il tutor risponde alla richiesta del broker comunicandogli che la musing è iniziata. Ciò permette al broker di tenere traccia di quali musing siano in esecuzione.

Il secondo metodo (**terminazione di una musing**) viene utilizzato dal broker nel caso in cui lo stato di gTopLevel cambi prima che tutte le musing relative a quello stato siano terminate. In questo caso il tutor è responsabile di terminare i thread associati alle musing come richiesto dal broker[5].

I tutor svolgono inoltre anche funzioni di client nei confronti del servizio di notifica dell'esito delle musing offerto dal broker. In particolare ogni thread associato ad una musing può terminare con successo (nel caso in cui l'applicazione della tattica sia andato a buon fine) o con un fallimento (nel caso in cui l'applicazione della tattica sollevi una eccezione). Nel primo caso il thread segnala al broker il successo includendo l'identificatore univoco della musing in esecuzione ed eventuali parametri aggiuntivi passati alla tattica; nel secondo caso al broker

[5]le implementazioni dei tutor di esempio in realtà non terminano i thread associati alle musing in quanto la libreria standard OCaml non implementa la funzione `Thread.kill`, ma si limitano ad ignorare il loro valore di ritorno quando questo sarà disponibile

viene segnalato il fallimento includendo unicamente l'identificatore della musing fallita.

Ogni tattica implementata nel proof assistant di HELM è una funzione avente il seguente prototipo[6]:

```
type tactic = status:(proof * goal) -> proof * goal list
```

Ogni tutor applica la funzione associata alla tattica a lui preposta. Se la funzione ritorna un valore l'applicazione della tattica è andata a buon fine e un successo può essere notificato al broker (è importante osservare che il valore di ritorno della tattica non è "interessante" per i tutor dato che non fanno parte del suggerimento). La tattica può anche non ritornare nessun valore sollevando l'eccezione `Fail`:

```
exception Fail of string
```

In questo caso l'applicazione della tattica non è andata a buon fine ed un fallimento viene notificato al broker.

Descrizione dei moduli

I moduli che implementano un tutor cambiano da tutor a tutor. Tuttavia un modulo predefinito viene fornito dall'implementazione di HBugs per la creazione di altri tutor: `Hbugs_tutors_common`.

Al suo interno sono implementate: una funzione per l'inizializzazione di un generico tutor, le funzioni per registrare e annullare la registrazione del tutor presso il broker e la funzione `load_state` responsabile della deserializzazione di uno stato contenuto in un messaggio di notifica del cambiamento di stato di gTopLevel.

Generazione automatica di tutor

L'implementazione di gTopLevel dispone di un certo insieme di tattiche già sviluppate avente una interfaccia comune (il prototipo `tactic` visto in sez. 7.2.4).

La maggior parte dei tutor implementabili corrisponderebbe ad implementazioni che condividono gran parte di codice, in particolare azioni quali la gestione delle richieste HTTP, la registrazione presso il broker, la creazione e la terminazione dei thread corrispondenti ad ogni musing risulterebbero identiche.

[6] le tattiche che richiedono argomenti sono formalizzate come funzioni che prendono tali argomenti e restituiscono una funzione avente tipo *tactic*

Se ci restringiamo al caso di tattiche che non richiedono argomenti aggiuntivi possiamo quindi realizzare una componente software che, data una funzione di tipo tactic ed un valore di tipo hint restituisce (sez. 7.5) restituisce l'implementazione di un tutor compatibile con HBugs che implementi un suggeritore per quella tattica.

Abbiamo quindi implementato un funtore ([LDG+02]) avente come modulo di input un modulo rappresentante una descrizione del tutor da implementare e come output un modulo implementante il corrispondente tutor.

La descrizione del tutor, riportata in fig. 7.7, richiede cinque argomenti: indirizzo IP e porta TCP sulla quale il tutor resterà in ascolto per richieste da parte del broker (righe 3–4), la tattica da utilizzare per tentare di procedere nella prova (riga 5), il suggerimento da restituire nel caso in cui la tattica vada a buon fine e una descrizione testuale delle funzionalità del tutor. La scelta di utilizzare funtori permette di disporre di tutti i controlli di type checking statico offerti dal compilatore OCaml.

```
1 module type HbugsTutorDescription =
2   sig
3     val addr: string
4     val port: int
5     val tactic: ProofEngineTypes.tactic
6     val hint: hint
7     val description: string
8   end
```

Figura 7.7: HBugs: input del funtore di generazione dei tutor

Il modulo restituito dal funtore espone una sola funzione:

```
val start:  unit -> unit
```

utilizzabile per iniziare l'esecuzione del tutor. L'implementazione di questo metodo si preoccupa di registrarlo presso il broker nella fase iniziale, di annullare questa registrazione prima di terminare e di gestire tutto le fasi di dialogo con il broker e di computazioni dei suggerimenti.

L'implementazione di un tutor in grado di effettuare suggerimenti abbinati ad una tattica che non richiede argomenti si riduce, utilizzando questo funtore, a poco più di una decina di righe di codice. Un tipica implementazione è riportata in fig. 7.8.

Ogni possibile implementazione di tutor dello stesso tipo prevede la scrittura di codice analogo a quello visto. Abbiamo quindi scelto di fattorizzare

```
module TutorDescription =
  struct
    let addr = "0.0.0.0"
    let port = 50001
    let tactic = Ring.ring_tac
    let hint = Hbugs_types.Use_ring_Luke
    let description = "Ring tutor, solve equalities on real numbers"
  end
;;
module Tutor = Hbugs_tutors_common.BuildTutor (TutorDescription) ;;
Tutor.start () ;;
```

Figura 7.8: HBugs: implementazione del tutor Ring

le informazioni necessarie alla implementazione di un tutor in una rappresentazione XML esterna e di realizzare un semplice applicativo che, data questa rappresentazione, generi sorgenti OCaml che implementino i tutor ivi descritti. Il codice rappresentato in fig. 7.8 è in effetti generato automaticamente da questo applicativo a partire dalla rappresentazione XML di fig. 7.9.

```
<tutor source="ring_tutor.ml">
  <addr>127.0.0.1</addr>
  <port>50001</port>
  <tactic>Ring.ring_tac</tactic>
  <hint>Hbugs_types.Use_ring_Luke</hint>
  <description>Ring tutor, solve equalities on real numbers</description>
</tutor>
```

Figura 7.9: HBugs: descrizione XML di un tutor

La rappresentazione XML contiene una sola informazione aggiuntiva rispetto ai parametri di input del funtore, ovvero il nome del file sorgente che implementa il tutor corrispondente utilizzato come destinazione della generazione di codice.

Le rappresentazioni XML dei vari tutor da generare automaticamente sono contenute in un unico file XML, detto *indice*, che contiene le rappresentazioni di tutti i tutor attualmente implementati. Include cioè una descrizione XML anche dei tutor il cui codice non è stato generato automaticamente. La disponibilità di questo indice ci ha permesso di realizzare semplicemente altri tool quali, ad esempio, gli script utilizzati per controllare l'esecuzione di tutti i tutor disponibili.

Un possibile sviluppo futuro correlato a questo indice, prevede la generazione del codice di un unico servizio web in grado di svolgere da solo il compito di

tutti i tutor implementati, minimizzando le richieste in termini di memoria e di porte TCP occupate. L'utilizzo di questo servizio risulterebbe particolarmente vantaggioso per esecuzioni standalone di gTopLevel.

7.2.5 gTopLevel (HBugs client)

gTopLevel è stato modificato per supportare l'interazione con HBugs all'interno del quale svolge ruolo di client. In particolare è state creata una componente software chiamata *Hbugs_client* che implementa le funzionalità necessarie affinché gTopLevel possa comunicare con HBugs e un'interfaccia grafica utilizzata per permettere all'utente un controllo interattivo di HBugs.

Il client hbugs è implementato nella classe `hbugsClient`.

Analogamente a quanto visto per broker e tutor, l'interazione di hbugsClient con HBugs è divisibile in un servizio web che attende richieste dal broker ed in una parte attiva che effettua richieste al broker. Inoltre hbugsClient dispone anche di una interfaccia formata dai metodi invocabili sulle sue istanze, utilizzata dall'implementazione di gTopLevel per interagire con HBugs.

L'architettura di hbugsClient e la sua interazione con gTopLevel possono essere schematizzate come in fig. 7.10.

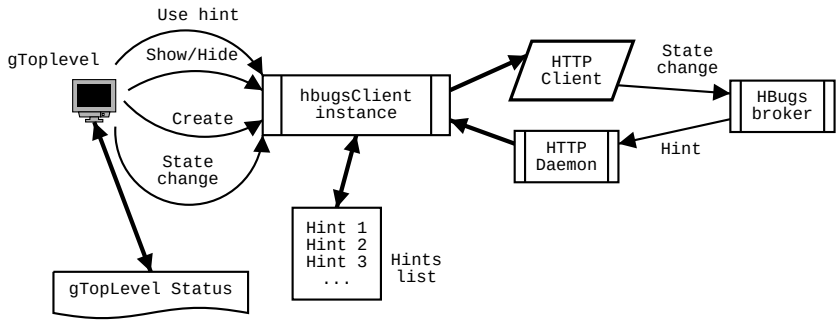


Figura 7.10: Hbugs_client: architettura

Il servizio web offerto da hbugsClient prevede la possibilità di invocare un unico metodo (Hint) utilizzato dal broker per segnalare la ricezione di un suggerimento da parte di un tutor. Alla ricezione di questa richiesta hbugsClient provvede ad aggiungere il suggerimento ricevuto alla lista dei suggerimenti attualmente applicabili. Questa lista ha inoltre una contro parte nell'interfaccia grafica di gTopLevel in modo che l'utente posso accorgersi dell'arrivo di un nuovo suggerimento.

La parte attiva di hbugsClient viene utilizzata per notificare al broker un

cambiamento di stato di gTopLevel in modo che questi possa attivare i tutor necessari. Il messaggio inviato da hbugsClient al broker contiene una serializzazione del nuovo stato del proof assistant.

L'istanziazione di un nuovo oggetto hbugsClient è *lazy*; viene cioè eseguita solamente la prima volta che il supporto per HBugs viene abilitato dall'utente di gTopLevel. Come effetto collaterale dell'istanziazione, hbugsClient provvede a registrarsi presso il broker come descritto in sez. 7.2.3.

La fase di sottoscrizione ai tutor attualmente disponibili è interamente gestita dall'interfaccia grafica di hbugsClient. Tale interfaccia è inoltre controllabile da due metodi che permettono di mostrarla e nasconderla all'utente.

Infine hbugsClient mette a disposizione di gTopLevel due metodi. Il primo, *stateChange*, viene utilizzato per notificare al broker un cambiamento di stato. stateChange si occupa di serializzare lo stato attuale di gTopLevel e di inviarlo al broker. Il secondo, *hint*, permette di accedere ai suggerimenti contenuti nella hints list.

7.3 Messaggi

Tutti i messaggi scambiati tra i diversi attori di HBugs sono definiti nel tipo concreto `message` del modulo Hbugs_common.

Ad ognuno di essi corrisponde una rappresentazione XML che può essere ottenuta serializzando un valore di tipo message utilizzando la funzione Hbugs_messages.string_of_msg.

```
val string_of_msg:  Hbugs_types.message -> string
```

Dualmente è possibile deserializzare un valore di tipo message da una sua rappresentazione XML utilizzando la funzione Hbugs_message.msg_of_string.

```
val msg_of_string:  string -> Hbugs_types.message
```

Tutte le interazioni tra attori di HBugs si basano sul modello di scambio di messaggi *richiesta/risposta* dove ogni richiesta ed ogni risposta sono valori di tipo message.

Il mittente di un messaggio lo serializza nel suo formato XML ed invia una richiesta utilizzando il metodo POST di HTTP al server HTTP del destinatario (ogni attore di HBugs dispone infatti di un server HTTP di cui è noto l'URL) contenente come corpo la serializzazione XML della richiesta. Il ricevente analizza la richiesta ricevuta e deserializza un valore di tipo message dal contenuto

della stessa, elabora poi un messaggio di risposta (sempre avente tipo message). Il messaggio di risposta viene serializzato in XML e inviato come contenuto della risposta HTTP al mittente originale. Il mittente analizza ora il contenuto della risposta HTTP ricevuta e deserializza la risposta al suo messaggio originale dal corpo della stessa.

Il modulo Hbugs_common mette a disposizione degli attori di HBugs una funzione che implementa lo scambio di messaggi appena descritto permettendo all'implementatore di lavorare unicamente su valori di tipo message:

```
val submit_req:  url:string -> Hbugs_types.message -> Hbugs_types.message
```

Il parametro *url* rappresenta l'URL di accesso al server HTTP del destinatario del messaggio. Il secondo parametro rappresenta il messaggio da inviare. Il valore di ritorno rappresenta il messaggio ricevuto in risposta.

I messaggi scambiati tra i vari attori di HBugs (corrispondenti ai possibili valori assunti dal tipo variante message) sono classificabili in cinque diverse categorie:

client → broker messaggi inviati da gTopLevel al broker (tab. 7.1)

tutor → broker messaggi inviati dai tutor al broker (tab. 7.2)

broker → client messaggi inviati dal broker a gTopLevel (tab. 7.3)

broker → tutor messaggi inviati dal broker ai tutor (tab. 7.4)

generici messaggi generici che possono essere scambiati tra ogni coppia di attori (tab. 7.5)

In ognuna delle tabelle indicate, il campo *Nome messaggio* corrisponde al nome del costruttore di tipo variante message.

7.4 Serializzazione dello stato

7.4.1 Stato di gTopLevel

Una rappresentazione schematica dello stato interno di gTopLevel durante il processo di dimostrazione è riportata in fig. 7.11.

Ogni dimostrazione in corso (*proof*) all'interno di gTopLevel è rappresentata da una tupla di quattro elementi.

Il primo elemento (**uri**) è la URI che identifica la dimostrazione corrente, viene impostata dall'utente all'inizio del processo di dimostrazione e sarà la

Nome messaggio	**Parametri**	**Descrizione**
Register_client	client_id, client_url	richiesta di registrazione di gTopLevel presso il broker. Il primo argomento verrà conservato per successive autenticazioni, il secondo per contattare gTopLevel
Unregister_client	client_id	richiesta di annullamento di una precedente registrazione di gToplevel presso il broker. Il parametro viene utilizzato per l'autenticazione del messaggio e per verificare l'esistenza di una precedente registrazione
List_tutors	client_id	richiesta della lista dei tutor attualmente registrati presso il broker. Il parametro viene utilizzato per l'autenticazione del messaggio
Subscribe	client_id, tutor_id list	richiesta di sottoscrizione ad un insieme di tutor attualmente registrati presso il broker. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo è una rappresentazione dell'insieme dei tutor ai quali si richiede la sottoscrizione
State_change	client_id, state	notifica al broker del cambiamento dello stato del proof assistant. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo contiene una rappresentazione del nuovo stato del proof assistant
Wow	client_id	risposta alla ricezione di un nuovo suggerimento da parte del broker. Il parametro viene utilizzato per l'autenticazione del messaggio

Tabella 7.1: HBugs: messaggi client → broker

stessa che identificherà la dimostrazione nel caso in cui l'utente decida di salvarla all'interno della libreria di HELM.

Il secondo elemento (**metasenv**) è una lista di *goal* aperti, che necessitano di essere dimostrati per poter concludere la dimostrazione. Ognuno di essi è identificato da un indice numerico intero, ha un contesto (*meta*) e un tipo (*goal*) corrispondente a ciò che deve dimostrare. Il contesto è formato da una lista di ipotesi, ad ognuna delle quali è associato un nome; ogni ipotesi può essere una *dichiarazione*, una *definizione* oppure *None* nel caso in cui non sia più accessibile in quel contesto.

Il terzo elemento della tupla (**(in)complete proof**) è la prova attuale eventualmente incompleta, ovvero contenente metavariabili corrispondenti a termini

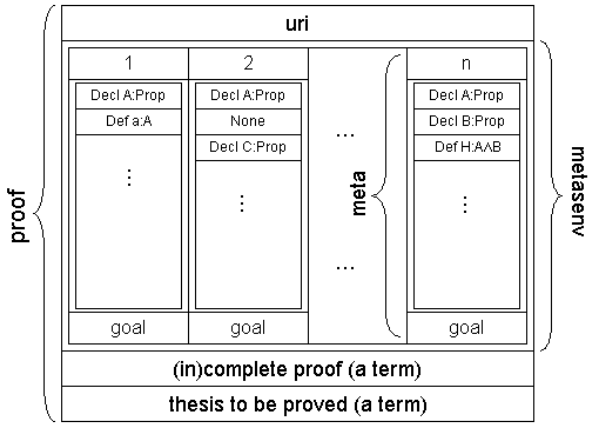


Figura 7.11: gTopLevel: stato

non ancora dimostrati.

Il quarto elemento della tupla (**term to be proved**) è il termine che si sta cercando di dimostrare.

Lo stato di gTopLevel (*status*) è formato da una dimostrazione in corso associata ad un identificatore numerico che referenzia un goal contenuto in metasenv. Questo identificatore ccorrisponde al goal corrente visualizzato dall'utente e prende il nome di *goal*.

7.4.2 Serializzazione

Dal punto di vista implementativo, lo stato del proof assistant di HELM è definito come una coppia `proof * goal` dove goal è un intero e proof è definito come segue:

```
type proof = UriManager.uri * Cic.metasenv * Cic.term * Cic.term
```

Le componenti del tipo proof sono rispettivamente: la URI del teorema che l'utente sta dimostrando, la lista delle metavariabili utilizzate nella dimostrazione incompleta, un termine che rappresenta la dimostrazione (eventualmente incompleta) e un termine che rappresenta il tipo di ciò che si vuole dimostrare.

La serializzazione di questo tipo di dato viene effettuata generando una rappresentazione XML divisa in tre componenti fondamentali: il goal corrente, la prova incompleta, il termine che deve essere provato.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2
```

```
<!DOCTYPE gTopLevelStatus SYSTEM "...">

<gTopLevelStatus>

  <CurrentGoal><!-- ... --></CurrentGoal>

  <ConstantType name="..." params="..." id="...">
     <!-- ... -->
  </ConstantType>

    <!-- content of gTopLevel's "currentproof" file -->

  <CurrentProof of="..." id="...">
     <!-- ... -->
  </CurrentProof>

</gTopLevelStatus>
```

L'elemento ⟨CurrentGoal⟩ contiene una rappresentazione testuale dell'interno indicante il goal corrente. L'elemento ⟨ConstantType⟩ contiene una rappresentazione XML del termine Cic da provare. L'elemento ⟨CurrentProof⟩ contiene una rappresentazione XML della dimostrazione, eventualmente incompleta, corrente.

Per una descrizione più dettagliata del contenuto di ConstantType e CurrentProof si rimanda a [Sac00].

7.5 Serializzazione dei suggerimenti

L'insieme dei suggerimenti supportati da HBugs è destinato a cambiare nel tempo. È infatti possibile, se non addirittura auspicabile, che ogni nuovo tutor implementato offra la possibilità di indicare all'utente una nuova tipologia di suggerimenti. L'architettura dei suggerimenti di HBugs è stata quindi pensata per minimizzare le modifiche necessarie al codice per supportarne di nuove.

La serializzazione dei suggerimenti è stata realizzata analogamente a quanto già visto per la serializzazione dello stato: è stato definito un tipo di dato OCaml che rappresenta i valori di tipo "suggerimento" ed una coppia di funzioni utilizzate per serializzare e deserializzare valori di questo tipo in rappresentazioni XML.

Il tipo di dato OCaml, contenente gli attuali suggerimenti supportati da HBugs, e' definito nel modulo Hbugs_types come segue:

```
1  type hint =
2    | Use_ring_Luke
3    | Use_fourier_Luke
4    | Use_reflexivity_Luke
5    | Use_symmetry_Luke
6    | Use_assumption_Luke
7    | Use_contradiction_Luke
8    | Use_exists_Luke
9    | Use_split_Luke
10   | Use_left_Luke
11   | Use_right_Luke
12   | Use_apply_Luke of string
13   | Hints of hint list
```

I costruttori di tipo senza argomento (righe 2–11) rappresentano i suggerimenti che indicano all'utente di applicare tattiche che non richiedono argomenti aggiuntivi. In questa categoria ricadono ad esempio le tattiche capaci di concludere interamente goal, di riscriverli in altri termini o che procedono nella prova applicando costruttori di tipi induttivi.

Attualmente questi suggerimenti corrispondono ai tutor generati automaticamente (sez. 7.2.4). Questa corrispondenza non è però strettamente necessaria.

Il costruttore di tipo `Use_apply_Luke` (riga 12) indica a gTopLevel di applicare la tattica *Apply* che applica un termine per procedere nella prova. L'argomento del costruttore di tipo è una rappresentazione testuale del termine da applicare.

L'ultimo costruttore di tipo, `Hints` (riga 13), viene utilizzato per contenere ulteriori suggerimenti. Il suo utilizzo è necessario nel caso di tutor che siano in grado di restituire più di un suggerimento all'utente[7].

Ad ognuno dei costruttori del tipo hint è associata una rappresentazione XML. È risultato naturale associare elementi XML aventi *content type EMPTY* ([W3Ca]) per i costruttori di tipo privi di argomento, un elemento XML con contenuto testuale pari al termine da applicare per il suggerimento `Apply` e un elemento XML avente contenuto pari a una sequenza di rappresentazione XML di suggerimenti per `Hints`. Il DTD associato a questa rappresentazione è il seguente:

```
<!ENTITY % Hints "(
  ring | fourier | reflexivity | symmetry | assumption |
```

[7] È ad esempio il caso del tutor *search pattern apply* che effettua una ricerca all'interno della libreria per tutti i teoremi applicabili al goal corrente e restituisce una lista di suggerimenti apply per ognuno dei risultati ottenuti

```
  contradiction | exists | split | left | right |
  apply |
  hints
)">

<!ELEMENT hint (%Hints;)>

<!ELEMENT ring EMPTY>
<!ELEMENT fourier EMPTY>
<!ELEMENT reflexivity EMPTY>
<!ELEMENT symmetry EMPTY>
<!ELEMENT assumption  EMPTY>
<!ELEMENT contradiction EMPTY>
<!ELEMENT exists EMPTY>
<!ELEMENT split EMPTY>
<!ELEMENT left EMPTY>
<!ELEMENT right  EMPTY>

<!ELEMENT apply (#PCDATA)>

<!ELEMENT hints (%Hints;)+>
```

Le funzioni di serializzazione da tipo hint ad una rappresentazione XML e di corrispondente deserializzazione non sono esposte nei moduli di HBugs in quanto utilizzate implicitamente dalle funzioni di serializzazione e deserializzazione dei messaggi.

La parte di gTopLevel responsabile della gestione dei suggerimenti è una semplice funzione che può direttamente effettuare pattern matching sul tipo di suggerimento ricevuto e comportarsi di conseguenza essendo completamente disaccoppiata dalla rappresentazione XML che viene utilizzata per la comunicazione tra tutor e gTopLevel attraverso il broker.

La scelta di questa metodologia implementativa abbinata ai controlli di esaustività del compilatore OCaml sui pattern matching permette di estendere facilmente l'insieme dei suggerimenti supportati da HBugs.

7.6 Implementazione

HBugs dal punto di vista implementativo è stato diviso in quattro parti: una parte comune (*common*), il *client* GTK, il *broker* ed i *tutor*.

La parte common è divisa in sei moduli e consta di circa 1000 righe di codice OCaml. La sua implementazione ha richiesto circa 80 ore/uomo.

Il client è diviso in due moduli più un terzo generato automaticamene a partire dall'interfaccia grafica realizzata utilizzando Glade. L'implementazione dei primi due moduli consta di circa 650 righe di codice ed è stata realizzata in circa 40 ore/uomo di lavoro.

Il broker è diviso in due moduli e consta di circa 700 righe di codice OCaml. È stato realizzato in circa 40 ore/uomo di lavoro.

Il codice sorgente dei tutor (ad eccezion fatta di searchPatternApply, in corso di realizzazione) viene generato automaticamente a partire da una descrizione XML del funzionamento del tutor. La generazione automatica utilizza due moduli OCaml ed un template. Il generatore è esso stesso un script OCaml. L'insieme dei sorgenti necessari alla generazione dei tutor consta di circa 550 righe di codice OCaml ed ha richiesto 30 ore/uomo per la sua implementazione.

Nome messaggio	**Parametri**	**Descrizione**
Register_tutor	tutor_id, tutor_url, hint_type, description	richiesta di registrazione di un tutor presso il broker. Il primo parametro viene memorizzato dal broker per successive autenticazioni. Il secondo è l'URL al quale è possibile contattare il tutor. Il terzo è una rappresentazione del tipo di suggerimento che il tutor può fornire. Il quarto è una descrizione testuale delle funzionalità del tutor
Unregister_tutor	tutor_id	richiesta di annullamento di una precedente registrazione presso il broker. Il parametro viene utilizzato per l'autenticazione del messaggio e per il controllo dell'esistenza di una precedente registrazione
Musing_started	tutor_id, musing_id	risposta ad una richiesta di inizio di una nuova musing da parte del broker. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo per la registrazione nel musings registry dell'inizio di una nuova musing
Musing_aborted	tutor_id, musing_id	risposta ad una richiesta di terminazione di una precedente musing da parte del broker. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo per la rimozione della musing dal musing registry
Musing_completed	tutor_id, musing_id, musing_result	notifica al broker della terminazione di una musing. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo per identificare la musing nel musing registry. Il terzo contiene una rappresentazione dell'esito della musing

Tabella 7.2: HBugs: messaggi tutor → broker

Nome messaggio	**Parametri**	**Descrizione**
Client_registered	broker_id	risposta alla richiesta di registrazione di un client. Il parametro viene memorizzato per la successiva autenticazione dei messaggi ricevuti dal broker
Client_unregistered	broker_id	risposta alla richiesta di annullamento di una precedente registrazione. Il parametro viene utilizzato per l'autenticazione del messaggio
Tutor_list	broker_id, tutor_dsc list	risposta ad una richiesta della lista di tutor attualmente registrati. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo contiene una rappresentazione della lista di tutor attualmente registrati comprendente l'URL ed una descrizione testuale delle funzionalità di ognuno di essi
Subscribed	broker_id, tutor_id list	risposta ad una richiesta di sottoscrizione da parte di un client. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo contiene una rappresentazione dell'insieme di tutor ai quali il client si è appena registrato
State_accepted	broker_id, musing_id list, musing_id list	risposta ad un messaggio di notifica del cambiamento di stato da parte di gTopLevel. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo ed il terzo contengono una rappresentazione di una lista di musing id corrispondenti alla lista di musing terminata in quanto relative allo stato precedente e a quella delle musing iniziate relativamente al nuovo stato
Hint	broker_id, hint	notifica a gTopLevel della presenza di un nuovo suggerimento. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo contiene una rappresentazione del nuovo suggerimento comprensivo di eventuali parametri aggiuntivi necessari al suo utilizzo

Tabella 7.3: HBugs: messaggi broker → client

Nome messaggio	**Parametri**	**Descrizione**
Tutor_registered	broker_id	risposta alla richiesta di registrazione di un tutor presso il broker. Il parametro viene utilizzato per l'autenticazione del messaggio
Tutor_unregistered	broker_id	risposto alla richiesta di annullamento di una precedente registrazione di un tutor presso il broker. Il parametro viene utilizzato per l'autenticazione del messaggio e per la verifica dell'esistenza di una precedente registrazione
Start_musing	broker_id, state	richiesta di attivazione di una nuova musing. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo contiene una rappresentazione dello stato del proof assistant sul quale attivare la musing
Abort_musing	broker_id, musing_id	richiesta di terminazione di una musing attualmente in corso. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo indica al tutor quale musing debba essere terminata
Thanks	broker_id, musing_id	risposta alla ricezione di un suggerimento da parte di un tutor. Il primo parametro viene utilizzato per l'autenticazione del messaggio. Il secondo per segnalare al tutor la musing di riferimento
Too_late	broker_id, musing_id	analogo a Thanks, ma utilizzando nel caso in cui venga ricevuto un suggerimento relativo ad una musing non più valida (relativa cioè ad uno stato differente dallo stato attuale del proof assistant)

Tabella 7.4: HBugs: messaggi broker → tutor

Nome messaggio	**Parametri**	**Descrizione**
Help		richiesta di informazioni riguardo ai metodi disponibili, utilizzato anche come *ping* per verificare la disponibilità di un certo attore di HBugs
Usage	usage_-string	risposta ad una richiesta di informazioni riguardo ai metodi disponibili. Il parametro contiene una rappresentazione testuale dei metodi disponibili
Exception	exc_name, exc_value	utilizzato come risposta generica nel caso in cui una richiesta non possa essere portata a termine con successo. I due parametri indicano, rispettivamente, il tipo di eccezione verificatasi ed un eventuale valore ad essa associato

Tabella 7.5: HBugs: messaggi generici

Capitolo 8

Conclusioni

I modelli di architetture software *client/server* e *document centric* si sono affermati negli anni come modelli potenti e adatti a risolvere problematiche di modularizzazione in applicazioni software di notevoli dimensioni.

Il modello client/server permette agli utilizzatori finali di sfruttare componenti software fisicamente lontane da loro e incentiva la formazione di comunità scientifiche indipendentemente dalla distanza.

Il modello document centric permette di dividere applicazioni software complesse in due componenti: un archivio di informazioni strutturate e un insieme di applicazioni tra loro indipendenti in grado di operare su tali informazioni.

Nell'ambito specifico delle librerie di conoscenza matematica formalizzata questi modelli sono purtroppo ancora poco diffusi. L'obiettivo del progetto HELM è la creazione di una libreria di questo tipo che sfrutti le potenzialità di questi modelli. Lo stato dell'arte di questo progetto mostra che entrambi i modelli sono efficaci per la gestione di librerie matematiche.

HELM dimostra inoltre che è possibile realizzare applicazioni software complesse, che si basino su queste librerie, dividendole in piccole componenti che interagiscano tra loro secondo interazioni client/server. Rendendo ognuna di queste applicazioni utilizzabile come servizio web, secondo una interfaccia ben definita, si massimizza l'accessibilità delle informazioni e di conseguenza l'utilità della libreria. Risulta inoltre semplificata l'implementazione di nuove funzionalità che si basino su quelle già implementate.

Nell'ambito di questa tesi abbiamo studiato ed effettivamente implementato molte delle componenti del progetto HELM. Abbiamo inoltre reso ognuna di esse accessibile come servizio web basato sul protocollo HTTP e implementato una libreria generica che semplifica l'implementazione di server HTTP nel linguaggio di programmazione OCaml.

8.1 Componenti implementate

8.1.1 OCamlHTTP

La necessità di realizzare numerosi servizi web basati sul protocollo HTTP ha imposto una riflessione su quale parte di codice sarebbe stata comune tra le implementazioni degli stessi.

Da questa analisi sono risultate due osservazioni:

1. l'implementazione del binding al protocollo HTTP dei vari servizi è comune a tutti i servizi web

2. per il linguaggio OCaml non era disponibile una libreria che permettesse di implementare "facilmente" demoni HTTP senza doversi preoccupare, ad esempio, di problematiche di rete quali gestione di socket o parsing di messaggi HTTP

Replicare il codice che implementa il binding dei servizi web al protocollo HTTP non sarebbe stata una scelta adeguata per la futura mantenibilità del codice degli stessi.

Per risolvere questa problematica abbiamo implementato la libreria OCamlHTTP.

Utilizzando questa libreria è possibile implementare demoni HTTP nel linguaggio di programmazione OCaml seguendo due possibili approcci di programmazione: imperativo e funzionale. Per i nostri scopi è risultato particolarmente utile l'approccio funzionale che ci ha permesso di implementare i servizi web di HELM implementando semplici callback che vengono automaticamente invocate ogni qual volta una richiesta HTTP ben formata viene ricevuta.

La libreria OCamlHTTP si occupa interamente delle problematiche di rete relative all'inzializzazione di un server HTTP, della formalizzazione dei messaggi (richieste e risposte) HTTP e dell'eventuale gestione di richieste concorrenti.

Quest'ultima caratteristica è risultata particolarmente utile nell'implementazione di servizi web che mantengono uno *stato* quali, ad esempio, UWOBO.

8.1.2 HTTP Getter

Uno dei principi fondamentali nello sviluppo di HELM è l'accessibilità delle informazioni. La scelta del modello di distribuzione dei documenti della libreria e la realizzazione delle componenti software che la implementano influisce notevolmente sull'accessibilità.

L'analisi delle caratteristiche dei documenti che compongono la libreria ha evidenziato che questi documenti sono immodificabili, che è auspicabile la loro replicazione e che i requisiti tecnologici per la loro pubblicazione devono essere minimi. L'immodificabilità dei documenti permette di utilizzare tecniche di caching che migliorino le prestazione. La replicazione è auspicabile sia per motivazioni prestazionali che di resistenza ai guasti. I requisiti per la loro pubblicazione, infine, devono essere minimi per massimizzare le possibilità di collaborazione nella creazione della libreria.

L'architettura software che più è risultata adatta a soddisfare questi requisiti è stata mutuata dall'architettura del sistema di distribuzione dei pacchetti software del sistema operativo Debian GNU/Linux: APT – Advanced Package Tool.

L'applicazione di questa architettura ad HELM prevede l'utilizzo di una componente software, chiamata HTTP Getter. In questa tesi presentiamo una reimplementazione nel linguaggio di programmazione OCaml del getter. Per una descrizione dettagliata delle differenze tra questa e le precedenti implementazioni si veda sez. 3.6.

Il getter è pensato per essere fisicamente "vicino" all'utente che vuole accedere alla libreria, idealmente in esecuzione sul suo computer o in un computer disponibile nella sua rete di appartenenza.

Il getter viene configurato indicando una lista di server (FTP, HTTP o semplici path in un file system distribuito) che dispongono di documenti della libreria. Uno dei metodi offerti dall'interfaccia di servizio web offerto dal getter (*update*) indica di contattare ognuno di questi server e di scaricare gli indici dei documenti di cui dispongono.

Utilizzando questi indici il getter crea una mappa che associa ad ogni nome logico di un documento (URN) la URL alla quale i documenti sono disponibili. Utilizzando poi altri metodi del getter (*getxml*, *getxslt*, *getalluris*, ecc.) l'utente può accedere ai documenti presenti nella libreria senza mai avere necessità di conoscere la locazione fisica degli stessi.

Questo livello di indirezione rende possibile la replicazione dei documenti (ad ogni URN può corrispondere più di una URL), politiche di caching (il getter può mantenere una copia locale di alcuni documenti) e lo spostamento fisico di documenti senza disservizi per l'utente finale.

I requisiti tecnologici per la pubblicazione di documenti si riducono, utilizzando il getter, alla disponibilità di uno spazio HTTP, FTP o NFS ed alla creazione di un semplice indice testuale che descriva quali documenti sono attualmente disponibili.

8.1.3 UWOBO

La scelta di una architettura document centric per il progetto HELM pone l'attenzione sui documenti che compongono la libreria. Questi documenti, sebbene siano in formato XML e quindi testuali, non sono direttamente fruibili dall'utente finale.

Per i fini di visualizzazione dei documenti della libreria è necessario trasformare questi documenti in formati maggiormenti fruibili quali HTML, XHTML o MathML (Presentation). I primi due formati risultano utilizzabili nel caso in cui l'utente stia visualizzando i documenti utilizzando un comune browser. MathML risulta invece utilizzabile per utenti che dispongano di browser o altre applicazioni software più specifiche per la visualizzazione di questo formato.

Data la mole della libreria non è consigliabile mantenere copie di tutti i documenti nei diversi formati. È invece auspicabile l'implementazione di una o più applicazioni in grado di convertire tra un formato e l'altro. All'interno del progetto HELM si è scelto di descrivere queste conversioni nel linguaggio XSLT.

La complessità delle trasformazioni in gioco e la scelta architetturale di dividerle in due macro fasi (CIC *rightarrow* MathML Content, MathML Content *rightarrow* MathML Presentation / HTML / XHTML) hanno imposto una strutturazione delle trasformazioni in applicazioni successive di fogli di stile XSLT.

Tra i processori XSLT attualmente disponibili non ve ne era nessuno che supportasse nativamente l'applicazione di fogli di stile "a catena". Abbiamo quindi implementato UWOBO. Una precedente implementazione di UWOBO era già stata realizzata nel linguaggio di programmazione Java, in questa tesi ne presentiamo una reimplementazione nel linguaggio di programmazione OCaml. Per un confronto tra queste implementazioni si veda sez. 4.6.

UWOBO è un servizio web che permette di applicare catene di trasformazioni XSLT ad un singolo documento di input. UWOBO espone un insieme di metodi utilizzati per precaricare fogli di stile XSLT ed associare ad ognuno di essi una chiave testuale. Espone inoltre il metodo *apply* che richiede l'applicazione di una catena di trasformazioni (specificata indicando una lista di chiavi che referenziano fogli di stile caricati in precedenza) ad un documento XML (specificato indicandone la URL).

UWOBO supporta il passaggio di parametri ai fogli di stile. I parametri sono passati come argomenti al metodo apply e istanziano i parametri specificati nei fogli stile utilizzando il costrutto XSLT ⟨xsl:param⟩. I parametri possono essere passati globalmente o localmente. I parametri globali sono passati indi-

stintamente a tutti i fogli di stile. I parametri locali sono passati solo ad alcuni fogli di stile (specificati indicandone la chiave).

UWOBO permette infine di specificare come argomenti del metodo apply le *proprietà di output* della catena di trasformazioni. Le proprietà supportate corrispondono agli attributi dell'elemento XSLT ⟨xsl:output⟩. È possibile quindi specificare uno dei tre metodi di output previsti dalla specifica XSLT (*xml*, *html*, *text*), il media type, l'encoding, l'indentazione, ecc.

8.1.4 drawGraph

La libreria matematica di HELM e, più in generale, le librerie distribuite assieme ai proof assistant sono tipicamente utilizzate per agevolare la dimostrazione di nuovi teoremi. È infatti possibile utilizzare termini già dimostrati all'interno della dimostrazione di altri.

Risulta quindi semplice definire due relazioni sull'insieme degli oggetti contenuti all'interno della libreria di HELM. La prima di queste è la relazione "dipende da" che associa ad ogni oggetto l'insieme degli oggetti da cui esso dipende. La seconda è la relazione inversa "è utilizzato da" che associa ad ogni oggetto l'insieme degli oggetti che dipendono da esso.

All'interno del progetto HELM, queste relazioni sono conservate sotto forma di metadati RDF, contenuti in un apposito database, appartenenti a due diversi schemi RDF: *helm:rdf:forward* (per la relazione "dipende da") e *helm:rdf:backward* (per la relazione "è utilizzato da").

Utilizzando questi metadati è possibile realizzare grafi di dipendenza per ogni oggetto contenuto all'interno della libreria. Data la mole di questi due grafi, un utilizzo pratico di essi richiede di non visualizzarli interamente, ma di mostrare solo la parte di essi "vicina" all'oggetto di interesse. Il concento di vicinanza si base sulla distanza indotta dalla lunghezza in numero di archi dei percorsi che uniscono due nodi.

Abbiamo implementato un servizio web, utilizzabile in combinazione con UWOBO, che permette di creare pagine XHTML contenenti i grafi di dipendenza (sotto forma di immagini GIF) degli oggetti contenuti nella libreria. Questo servizio web è stato denominato *drawGraph*. Utilizzando drawGraph è possibile generare sia grafi di dipendenza forward che backward.

L'implementazione presentata è una reimplementazione nel linguaggio di programmazione OCaml di un servizio web analogo, realizzato dal dott. Claudio Sacerdoti Coen nel linguaggio di programmazione Perl.

L'interfaccia di drawGraph prevede un semplice metodo che prende come in-

put una rappresentazione Graphviz di un grafo orientato e restituisce una pagina XHTML che referenzia una immagine GIF contenente il grafo corrispondente. Per la generazione della rappresentazione testuale viene utilizzato UWOBO che a sua volta accede ai metadati utilizzando il getter.

Per la visita del grafo di dipendenza è stato necessario implementare un ulteriore servizio web (*uriSetQueue*), responsabile della gestione di code di URI di lunghezza finite.

8.1.5 searchEngine

La mole della libreria di HELM (attualmente composta da più di 100,000 documenti XML) rende difficoltose la ricerche di oggetti che soddisfino le richieste dell'utente. Queste difficoltà sono inoltre accentuate per utenti che non abbiano conoscenza della strutturazione delle libreria dei sistemi software dai quali i documenti sono stati estratti. Ricordiamo infatti che tipicamente la locazione dei documenti nello spazio delle URI di HELM dipende dalla disposizione degli stessi nelle librerie dei sistemi software di origine.

Il primo approccio implementato nell'ambito del progetto HELM per la risoluzione di queste problematiche è stata la realizzazione di MathQL. MathQL è un linguaggio di query definito ed implementato appositamente per effettuare ricerche su metadati RDF relativi a librerie di conoscenza matematica.

L'implementazione di MathQL è stata realizzata nel linguaggio di programmazione OCaml ed utilizzata per aggiungere funzionalità di ricerca al proof assistant gTopLevel. Utilizzando gTopLevel è infatti possibile effettuare diverse tipologie di query sulla libreria di HELM. È ad esempio possibile cercare tutti i termini il cui tipo conclude il goal corrente della dimostrazione in corso.

Le funzionalità di MathQL fino ad ora non erano fruibili al di fuori di gTopLevel. Per superare questa limitazione abbiamo implementato un servizio web, denominato *searchEngine*, che permette di effettuare query a diversi livelli di astrazione. Unico requisito per il suo utilizzo è disporre di un semplice browser HTML. searchEngine è stato pensato per essere utilizzato congiuntamente ad una interfaccia composta da un insieme di pagine HTML abbinate a Javascript ancora in fase di realizzazione.

searchEngine presenta un metodo (*execute*) che permette di eseguire direttamente query MathQL. L'uso di questo metodo è consigliato ai conoscitori del linguaggio di query MathQL. Per facilitare la creazione di query l'interfaccia web è in grado di seguire l'utente lungo le produzioni della grammatica di MathQL.

searchEngine permette poi di effettuare due query a più alto livello di astra-

zione: *locate* e *searchPattern*. La prima permette di trovare tutti i termini corrispondenti ad un identificatore testuale inserito dall'utente. La ricerca dell'identificatore "list" ad esempio restituisce un insieme di termini contenenti diverse definizioni di liste presenti all'interno della libreria.

La seconda permette di cercare tutti i termini aventi tipo specificato dall'utente. Il tipo può anche essere un pattern su tipi, ovvero non essere completamente istanziati. Questa caratteristica permette di cercare ad esempio tutti i termini che dimostrano che per ogni naturale n, n è uguale al prodotto di un naturale non specificato per n stesso.

Le query che permettono all'utente di specificare termini, attualmente solamente searchPattern, dispongono inoltre di un meccanismo di disambiguazione. Utilizzando la disambiguazione è permesso all'utente di non specificare le URI di tutti i termini a cui fa riferimento, ma solamente un loro identificatore. Utilizzando congiuntamente la query locate ed il proof checking è possibile disambiguare termini ambigui in maniera semi automatica.

8.1.6 HBugs

L'approccio alla dimostrazione interattiva dei proof assistant moderni simili a Coq prevede che l'utente fornisca uno *script* la cui esecuzione dimostri il goal. Lo script descrive una sequenza di applicazioni di tattiche. Lo script include anche eventuali argomenti necessari alla applicazione di una tattica; la tattica *Apply* ad esempio richiede come argomento il termine da applicare al goal corrente.

gTopLevel, il proof assistant realizzato nell'ambito del progetto HELM, non fa eccezione: all'utente è richiesto di fornire uno script. La sola differenza "pratica" rispetto a Coq è che lo script viene fornito utilizzando una interfaccia grafica piuttosto che una testuale[1].

L'approccio a script non fornisce molti aiuti all'utente. Questa limitazione risulta particolarmente svantaggiosa nel caso di utilizzo di tattiche dispendiose in termini di tempo. È ad esempio il caso di tattiche riflessive quali Ring o Fourier. La loro applicazione richiede una quantità di tempo sostanzioso per effettuare il type checking di tutti i termini da cui dipendono. L'utente può quindi dover attendere anche alcuni minuti (è il caso di Ring) prima di avere notifica del fatto che la tattica *non* è applicabile.

Per risolvere queste problematiche abbiamo studiato ed implementato una architettura di *suggerimenti* per gTopLevel.

[1] questa differenza di interfaccia rende anche non disponibili all'utente i *tatticali*, più adatti ad una interfaccia testuale

L'architettura ddi HBugs si fonda sull'utilizzo di due diverse tipologie di servizi web. gToplevel interagisce con un unico servizio web denominato *broker*, il broker interagisce sia con gTopLevel che con uno o più *tutor*.

I tutor sono i servizi web responsabili della generazione dei suggerimenti per l'utente finale. Ricevono come input una rappresentazione XML di uno stato di gTopLevel e restituiscono o una segnalazione di insuccesso o una rappresentazione di un suggerimento (*hint*). gTopLevel è in grado di interpretare questi suggerimenti e di segnalare all'utente la disponibilità di uno o più di essi. L'utente può utilizzare i suggerimenti per progredire nella dimostrazione.

Il broker svolge il ruolo di intermediario tra gTopLevel e i tutor. Utilizzando il broker gTopLevel viene a conoscenza di quali tutor siano attualmente disponibili e permette all'utente di selezionare quali voglia utilizzare. La selezione dei tutor segue il modello delle sottoscrizioni tipico della rete Usenet. Ogni qual volta lo stato interno di gTopLevel cambia, ad esempio in seguito all'utilizzo di una tattica da parte dell'utente, il nuovo stato viene serializzato e inviato al broker. Questi provvede poi a inoltrare questo stato ai tutor selezionati dall'utente. I suggerimenti generati dai tutor vengono raccolti dal broker e inoltrati a gTopLevel.

Oltre all'infrastruttura, sono stati sviluppati anche una serie di semplici tutor corrispondenti a tattiche primitive (quali[2] Left, Right, Reflexivity, ecc.) e riflessive (quali Ring e Fourier). È inoltre in corso di sviluppo un tutor più sofisticato (*searchPatternApply*) in grado di cercare all'interno della libreria di HELM quali termini siano applicabili per concludere il goal corrente. Per ogni risultato di questa ricerca viene "suggerito" all'utente di utilizzare la tattica Apply con argomento corrispondente al termine trovato.

8.2 Sviluppi futuri

I possibili sviluppi futuri del lavoro di tesi svolto sono molteplici. Li riportiamo qui divisi per argomento di attinenza.

implementazione di servizi web intendiamo approfondire l'approccio proposto in sez. 1.2.4 per l'automatizzazione della creazione di servizi web basati su applicazioni software già esistenti.

In particolare riteniamo sia possibile generare, a partire da file di interfaccia .mli di moduli OCaml, sia un documento WSDL che una implementazione di un servizio web. Il documento WSDL corrisponde alla descrizione

[2] riportate utilizzando i nomi delle tattiche del proof assistant Coq corrispondenti

del servizio web generato nel linguaggio in corso di standardizzazione da parte del Working Group Web Services Description del W3C. L'implementazione corrisponde ad un demone HTTP in grado di implementare sia il binding HTTP che il binding SOAP del servizio web descritto nell'interfaccia WSDL.

I file .mli non contengono abbastanza informazioni per descrivere un servizio web come richiesto da WSDL. Riteniamo però sia possibile colmare questa deficienza utilizzando tecniche di literate programming.

Abbiamo già implementato un parser in grado di elaborare file di interfaccia .mli e commenti in esso contenuto opportunamente formattati contenenti informazioni addizionali necessarie a descrivere il servizio web.

Abbiamo anche già implementato alcune classi OCaml che formalizzano descrizioni WSDL. Tali classi non sono però risultate soddisfacenti; le recenti modifiche apportate a WSDL[3] hanno inoltre reso necessarie molte modifiche.

Stiamo attualmente studiando una migliore formalizzazione OCaml delle descrizioni WSDL.

Per quanto riguarda i binding è possibile utilizzare OCamlHTTP per implementare facilmente il binding HTTP. Per il binding SOAP stiamo studiando l'utilizzo di OCaml-Soap[4] che sembra però essere ad uno stato ancora primitivo; risulterà probabilmente necessario prendere parte al suo sviluppo.

OCamlHTTP la libreria implementata per la realizzazione di demoni HTTP (OCamlHTTP) è risultata promettente, ma è ancora ben lontata dall'essere considerata una libreria "matura".

Dal punto di vista del supporto per il protocollo HTTP intendiamo aggiungere il supporto per:

- i metodi definiti in [RFCb] (attualmente sono supportati solamente GET e POST)
- l'autenticazione Basic
- le connessioni persistenti
- il pipelining di richieste HTTP

[3] attualmente è disponibile solamente una Working Draft delle specifiche del linguaggio
[4] `http://caml.inria.fr/ocaml-soap/`

Dal punto di vista architetturale intendiamo invece rendere disponibile una variante della modalità di gestionde dei client *'Thread*. Questa variante prevede la disponibilità di un insieme finito di thread preallocati in modo da evitare l'overhead di creazione di nuovi thread per ogni richiesta HTTP.

Dal punto di vista della API infine sono da appianare alcune fastidiose incoerenze. È inoltre possibile convertire alcuni controlli che attualmente vengono effettuati a runtime in controlli statici effettuabili dal compilatore.

HBugs il suggeritore implementato per gTopLevel è probabilmente uno degli aspetti di questa tesi che lascia più spazio per futuri approfondimenti.

L'insieme dei tutor attualmente disponibili soffre di due problemi: è troppo esiguo ed implementa funzionalità triviali. Risulta quindi necessario implementare altri tutor dotati di maggiore "intelligenza" per testare appieno le potenzialità del sistema di suggerimenti. Un primo passo in questa direzione è indubbiamente il completamente dell'implementazione del tutor searchPatternApply.

L'implementazione dei tutor triviali, intendendo con questo termine tutor che suggeriscono l'uso di tattiche semplici quali l'applicazione di costruttori induttivi, non è comunque priva di utilità. Stiamo infatti considerando la possibilità di utilizzare questi tutor per disabilitare nell'interfaccia grafica di gTopLevel le funzionalità di tattiche che risultino non applicabili.

Dal punto di vista architetturale siamo soddisfatti del meccanismo di generazione automatica dei tutor a partire dall'indice XML (sez. 7.2.4). È possibile utilizzare lo stesso indice anche per altre finalità. Stiamo attualmente considerando la possibilità di generare a partire da questo indice un unico demone HTTP che supporti le funzionalità di tutti i tutor ivi descritti. Un simile demone risulterebbe particolarmente utile, in quanto poco dispendioso in termini di risorse, per esecuzioni standalone di gTopLevel.

Certamente il confronto tra HBugs e Omega Ants ([BS98]) merita maggiori considerazioni per capire quali delle soluzioni di questo sistema possano essere utilizzate nell'ambito del progetto HELM. Merita interessa lo studio di una *blackboard architecture* che, abbinata ad HBugs, permetta di automatizzare maggiormente il processo di dimostrazione, come descritto in [BS00].

Restano infine aperte molte problematiche di usabilità della interfaccia

grafica di gTopLeve e, in particolare, su come presentare all'utente i suggerimenti ricevuti dai tutor. La presentazione di questi suggerimenti deve risultare efficace, ma non eccessivamente invasiva per evitare di distrarre l'utente.

UWOBO sebbene l'implementazione OCaml di UWOBO sia risultata migliore della precedente sotto molto aspetti, restano alcune problematiche da risolvere.

Il maggiore problema riscontrato risiede nella assenza di feedback durante l'elaborazione dei fogli di stile all'atto del loro caricamento. Per risolvere questo problema sarà probabilmente necessario modificare l'implementazione del binding OCaml di libxslt che attualmente non espone le funzioni necessarie a definire gli handler invocati nel caso si verifichino errori.

Grafi di dipendenza l'attuale architettura del generatore di grafi di dipendenza risulta eccessivamente complessa.

Parte di questa complessità è da ricercarsi nella impossibilità di utilizzare direttamente dal linguaggio di programmazione OCaml le librerie di Graphviz.

Stiamo studiando l'implementazione di un binding OCaml di queste librerie che permetterebbe di rimuovere alcuni degli attori attualmente necessari per la creazione dei grafi di dipendenza.

Appendice A

WST: tester per servizi web

Nell'ambito di questa tesi sono stati sviluppati molti servizi web. Fin dallo sviluppo del primo di essi si è presentato il problema di come effettuare il debugging delle loro funzionalità. A differenza di normali applicazioni standalone infatti il debugging dei servizi web risulta più complesso in quanto le interazioni con essi non si svolgono su un singolo host e non interagiscono direttamente con le periferiche utilizzabili direttamente dall'utente quali mouse, tastiera, ecc.

Per quanto riguarda l'input, osserviamo che tutti i servizi web sviluppati si basano sul protocollo HTTP e sui metodi GET e POST. I parametri di input sono quindi codificati o nella URL di accesso al servizio o nel corpo della richiesta utilizzando la codifica *URL encoding* ([RFCb]). Per le richieste di tipo GET i parametri possono essere inseriti in un comune browser dovendosi però preoccupare di effettuare "manualmente" l'escaping dei caratteri non validi per le URL o non ammessi in un determinato contesto all'interno della URL. Per le richieste di tipo POST i problemi risultano ancora maggiori dato che nessun browser permette all'utente di effettuare direttamente richieste di questo tipo. Le soluzioni comunemente utilizzate per ovviare a questo problema prevedono la realizzazione di form HTML che utilizzino il metodo POST o l'utilizzo diretto di *telnet*.

Per quanto riguarda l'output osserviamo che questo è apprezzabile utilizzando un browser solamente se il formato di output è supportato da questi e se l'header HTTP Content-Type è impostato correttamente. Purtroppo però pochi browser supportano la visualizzazione di XML (a meno che non sia abbinato ad un foglio di stile CSS) e in generale non possiamo fare l'assunzione che il Content-Type sia impostato correttamente, dal momento che ci stiamo occupando di debugging. Anche nel caso in cui questi due requisiti siano soddisfatti, osserviamo che i browser raramente ci permettono di apprezzare caratteristiche

quali gli header della risposta HTTP ricevuta. La soluzione usuale per lo studio dell'output impone di utilizzare direttamente *telnet*.

L'uso di *telnet* per il dialogo diretto con servizi web risulta indubbiamente potente, ma anche difficoltoso (in quanto richiede la conoscenza sintattica del protocollo HTTP e delle convenzioni utilizzate dai browser più diffusi) e soprattutto difficilmente automatizzabile.

Per ovviare a queste problematiche abbiamo realizzato una semplice applicazione software utile per il debugging di servizi web paragonabile come utilità a telnet ma:

- facilmente automatizzabile
- in grado di gestire automaticamente URL escaping
- utilizzabile senza particolare conoscenza della sintassi del protocollo HTTP

L'applicazione realizzata prende il nome di *WST* (Web Service Tester), è una applicazione GTK[1] realizzata in Python[2] con l'ausilio di Glade[3] per la realizzazione dell'interfaccia grafica. Uno screenshot di WST è riportato in fig. A.1.

L'interfaccia grafica di WST è divisa in tre *frame*:

Location frame all'interno di questo frame deve essere specificata la URL del servizio web che si vuole testare, utilizzando il tasto *Test!* è inoltre possibile verificare se il servizio web specificato sia attualmente disponibile[4]

Query frame le funzionalità di questo frame permettono di specificare il tipo di richiesta da inviare al servizio web. In particolare è possibile specificare il path della richiesta, i parametri della stessa (sia per il nome che per il valore dei parametri, l'URL escaping viene eseguito automaticamente) ed il metodo utilizzato (GET o POST). I parametri vengono passati come *query string* per richieste GET e all'interno del corpo per richieste POST. Nel caso di richieste POST è inoltre possibile impostare manualmente il corpo della richiesta. Il tasto submit viene utilizzato per inviare la richiesta al servizio web

[1] `http://www.gtk.org`
[2] `http://www.python.org`
[3] `http://glade.gnome.org`
[4] l'implementazione di questa funzionalità si basa sull'assunzione, valida per tutti i servizi web implementati nell'ambito di questa tesi, che il servizio web supporti una richiesta GET avente path */help*

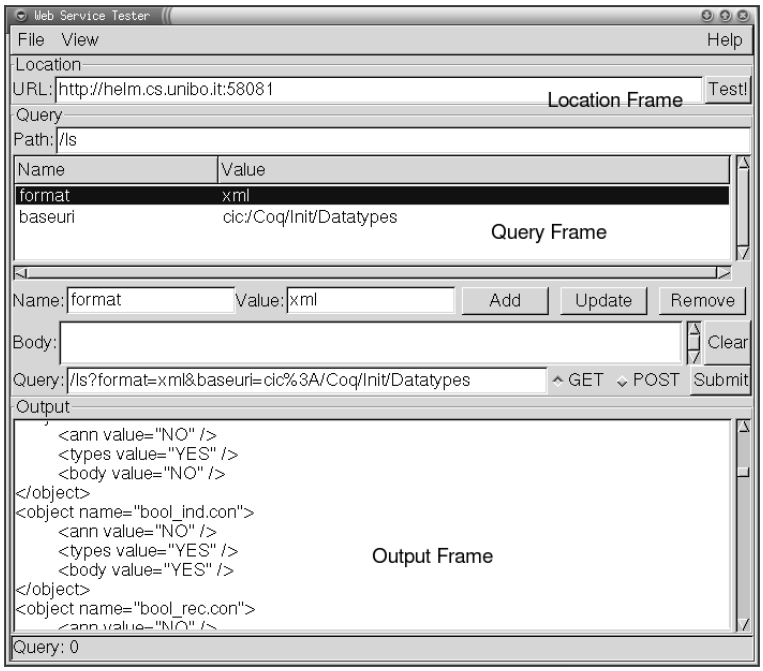


Figura A.1: WST (Web Service Tester)

Output frame all'interno di questo frame viene riportata la risposta HTTP del servizio web alla richiesta appena inviata

Per quanto riguarda l'automazione del processo di debugging, WST supporta *script* utilizzando i quali è possibile effettuare una serie di richieste HTTP in modalità *batch*. WST tiene traccia del risultato di ogni richiesta e permette all'utilizzatore di visionare ognuno di essi. Gli script sono inoltre caricabili e salvabili su disco. La loro codifica è una rappresentazione testuale di valori Python, corrispondenti alle singole richieste, serializzati utilizzando la funzione *repr* e deserializzati utilizzando la funzione *eval* di Python. L'utilizzo di script è risultato particolarmente utile per il debugging di servizi web aventi stato quali ad esempio UWOBO.

A.1 Implementazione

WST è stato realizzato in un singolo file sorgente Python in aggiunta ad alcuni script esterni utilizzati per includere nel sorgente stesso la rappresentazione Glade dell'interfaccia grafica.

L'implementazione consta di circa 400 righe di codice Python ed ha richiesto poco più di una giornata di lavoro (10 ore/uomo).

Bibliografia

[APS+] Asperti A., Padovani L., Sacerdoti Coen C., Guidi F., and Schena I. Mathematical knowledge management in helm. *Annals of Mathematics and Artificial Intelligence*.

[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. Towards a Library of Formal Mathematics. Accepted at TPHOLS 2000.

[BBC+97] Barras B., Boutin S., Cornes C., Courant J., Filliâtre J. C., Giménez E., Herbelin H., Huet G., Munoz C., Murthy C., Parent C., Paulin-Mohring C., Saibi A., and Werner B. The Coq Proof Assistant Reference Manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

[BC+] Benzmuller C., Cheikhrouhou L., et al. Omega: Towards a mathematical assistant.

[BS98] Benzmuller C. and Sorge V. A blackboard architecture for guiding interactive proofs, 1998.

[BS00] Benzmuller C. and Sorge V. An open approach at combining interactive and automated theorem proving, 2000.

[CP02a] Casarini P. and Padovani L. "Gnome DOM Engine". Web page, April 2002. `http://gdome2.cs.unibo.it/`.

[CP02b] Casarini P. and Padovani L. "The Gnome DOM Engine". *Markup Languages: Theory & Practice*, 3(2):173–190, April 2002.

[FM03] Filliâtre J. C. and Marché C. ocamlweb: a literate programming tool for objective caml. Technical report, Inria (Institut National de Recherche ed Informatique ed en Automatique), France, 2003.

[GKNK93] Gansner E. R., Koutsofios E., North S. C., and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[GN99] Gansner E. R. and North S. C. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 00(S1):1–5, 1999.

[GS03] Guidi F. and Schena I. A query language for a metadata framework about mathematical resources. In *Proceedings of MKM 2003 (Bertinoro, Italy, Feb 2003)*, 2003.

[Gui03] Guidi F. *Searching and Retrieving in Content-based Repositories of Formal Mathematical Knowledge*. PhD thesis, Università di Bologna, 2003.

[Knu92] Knuth D. E. *Literate Programming*. Center for the Study of Language and Information, 1992.

[LDG+02] Leroy X., Doligez D., Garrigue J., Didier R., and Vouillon J. The objective caml system release 3.06 documentation and user's manual, August 2002. http://caml.inria.fr/ocaml/htmlman/index.html.

[Ned03] Nediani A. Disegno e implementazione di un'interfaccia web di supporto ad interrogazioni su basi di dati documentarie. Master's thesis, Università di Bologna, 2003.

[OBS99] Olson M. A., Bostic K., and Seltzer M. Berkeley DB. Technical report, Sleepycat Software, Inc., 1999.

[pth96] Portable operating system interface (posix®) – part 1: System application program interface (api) [c language], 1996. Standard ISO/IEC 9945-1: 1996 (E) IEEE Std 1003.1.

[RFCa] HTTP Authentication: Basic and Digest Access Authentication Request for Comments: 2617 June 1999. http://www.ietf.org/rfc/rfc2617.txt.

[RFCb] Hypertext Transfer Protocol – HTTP/1.1 Request for Comments: 2616 June 1999. http://www.ietf.org/rfc/rfc2616.txt.

[RFCc] Uniform Resource Identifiers (URI): Generic Syntax Request for Comments: 2396 August 1998. http://www.ietf.org/rfc/rfc2396.txt.

[Sac00] Sacerdoti Coen C. Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata. Master's thesis, Università di Bologna, 2000.

[Sea02] Sean M. Burke. *Perl & LWP*. O'Reilly, June 2002.

[Ste96] Stevens W. R. Tcp for transactions, http, nntp, and the unix domain protocols, 1996.

[Vei03a] Veillard D. Libxml library reference, 2003. http://xmlsoft.org/html/libxml-lib.html.

[Vei03b] Veillard D. Libxslt library reference, 2003. http://xmlsoft.org/XSLT/html/libxslt-lib.html.

[W3Ca] Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation. http://www.w3.org/TR/2000/REC-xml-20001006.

[W3Cb] SOAP Version 1.2 Part 1: Messaging Framework W3C Candidate Recommendation. http://www.w3.org/TR/2002/CR-soap12-part1-20021219.

[W3Cc] SOAP Version 1.2 Part 2: Adjuncts W3C Candidate Recommendation. http://www.w3.org/TR/2002/CR-soap12-part2-20021219.

[W3Cd] Web Services Description Language (WSDL) Version 1.2 W3C Working Draft 24 Jan 2003. http://www.w3.org/TR/2003/WD-wsdl12-20030124.

[W3Ce] Web Services Glossary W3C Working Draft 14 Nov 2002. http://www.w3.org/TR/2002/WD-ws-gloss-20021114.

[W3Cf] XML Schema Part 0: Primer W3C Recommendation. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502.

[W3Cg] XML Schema Part 1: Structures W3C Recommendation. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502.

[W3Ch] XML Schema Part 2: Datatypes W3C Recommendation. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

[W3Ci] XSL Transformations (XSLT) Version 1.0 W3C Recommendation. http://www.w3.org/TR/1999/REC-xslt-19991116.

Ringraziamenti

Ringrazio il prof. **Andrea Asperti**, relatore di questa tesi, per la fiducia che ha riposto in me e per la sua capacità di mostrare sempre agli studenti *the big picture* di ciò che spiega.

Ringrazio il dott. **Claudio Sacerdoti Coen** (AKA "CSC"), corelatore di questa tesi, per la sua disponibilià. È sempre stato presente, nonostante i suoi molteplici impegni, per offrire spiegazioni, per confrontare idee, per discutere possibili implementazioni e per rileggere pagine e pagine di tesi. Mi ha inoltre offerto molti spunti per capire come "giri il mondo" universitario.

Ringrazio il dott. **Luca Padovani** per il confronto sia sul piano tecnico, che su quello umano. A sua insaputa, la sua capacità di (s)drammatizzare ed il suo modo di rapportarsi al lavoro mi hanno spronato più di una volta.

Ringrazio inoltre aggregatamente Andrea, Claudio e Luca: lo "zoccolo duro" del progetto **HELM**. Lavorare con loro è stato più che oggettivamente produttivo, è stato più che scientificamente stimolante ... è stato divertente!

Ringrazio il prof. **Renzo Davoli**, docente del corso di Sistemi Operativi, per avermi dato la possibilità di partecipare al progetto *CariStudenti*. Lavorando con lui e con gli altri ragazzi membri del progetto non solo ho acquisito molte competenze tecniche, ma ho anche conosciuto un modo di rapportarsi all'informatica che l'università non può insegnare.

Ringrazio infine i miei **compagni di viaggio** in questi cinque anni di vita universitaria: Leo, Galla, Iolanda, Andrea, Simona, Roberta, Samuele, Roberto, Luca, ... Ho condiviso con loro lezioni interessanti, lezioni noiose, successi ed insuccessi agli esami, progetti, molte cene ed almeno altrettante bevute. Sarà soprattutto grazie a loro se ricorderò per sempre con piacere questa avventura.