UNIVERSITÀ
DI TORINO

di.unito.it

DIPARTIMENTO
DI INFORMATICA

UNIVERSITÀ DI TORINO
DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA TRIENNALE IN INFORMATICA

# Editing di ontologie tramite il linguaggio di programmazione funzionale ℂDuce

**Relatore**
Professoressa Viviana Bono

**Laureando**
Davide Camino
Matricola: 897753

ANNO ACCADEMICO 2022-2023

Data di laurea 14 Novembre 2023

*A tutti quelli che mi vogliono bene.*

*L'informatica riguarda i computer*
*non più di quanto l'astronomia riguardi i telescopi.*
*E. W. Dijkstra*

*Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.*

# ABSTRACT

Questo lavoro illustra lo sviluppo di strumenti per l'editing di ontologie tramite il linguaggio di programmazione funzionale ℂDuce, in particolare la realizzazione di programmi per il refactoring, il merge e la traduzione da tesauri a ontologie. Lo scopo dello studio è quello di valutare se, e in che condizioni, lo sviluppo di strumenti più o meno ad hoc per l'editing di ontologie sia vantaggioso rispetto all'uso di strumenti grafici tradizionali come Protégé. Nella valutazione si tengono in considerazione principalmente la difficoltà tecnica e il tempo di sviluppo degli strumenti creati con ℂDuce e il tempo necessario e la ripetitività per fare le stesse modifiche con strumenti grafici.

# Indice

# Capitolo 1

# Concetti di base

## 1.1 Introduzione

Qui illustriamo alcuni dei concetti di base che serviranno per comprendere il resto della discussione, daremo una definizione di ontologia e tesauro, descriveremo brevemente gli strumenti utilizzati e i linguaggi con cui si descrivono le basi di conoscenza che tratteremo.

Questo capitolo non ha la pretesa di descrivere approfonditamente ogni dettaglio degli argomenti presentati, piuttosto quella di definire alcuni termini che ricorreranno frequentemente nell'elaborato, per permettere la lettura anche a chi non ha mai trattato con la rappresentazione formale della conoscenza.

## 1.2 Basi di conoscenza

Una base di conoscenza (Knowledge Base, KB) è un insieme di affermazioni, ognuna delle quali espressa attraverso un linguaggio di rappresentazione della conoscenza. Le affermazioni esprimono concetti riguardanti il dominio di interesse.

Perché una base di conoscenza possa essere utile, deve essere possibile aggiungere conoscenza (fare affermazioni) e interrogare la KB; queste operazioni possono coinvolgere meccanismi di inferenza che permettano di ricavare nuove affermazioni da quelle già note. L'inferenza sfrutta una logica formale per ricavare nuove informazioni partendo da quelle che sono già presenti nella KB [20].

### 1.2.1 Ontologie

#### Definizione

In accordo con una definizione ampiamente accettata [9], un'ontologia ha lo scopo di rappresentare un vocabolario per definire i concetti di un particolare dominio di interesse condiviso ed è costituita da definizioni di classi, relazioni, funzioni e altri oggetti utili a rappresentare la conoscenza [8].

La definizione è ancora un po' vaga: più precisamente, un'ontologia è una base di conoscenza che permette di descrivere concetti e relazioni tra di essi, specificando questi oggetti tramite un linguaggio di rappresentazione della conoscenza basato su una logica formale.

### Scopo delle ontologie

Nel contesto del web semantico[1], le ontologie sono il mezzo principale per condividere, integrare e scoprire dati [9].

### Manipolare ontologie

Dato lo scopo delle ontologie, la possibilità di riutilizzare, modificare e ampliare ontologie esistenti risulta essere un argomento centrale. Spesso le basi di conoscenza sono sì strutturate, ma sono eterogenee e non permettono interoperabilità. Scopo di questo lavoro è quello di presentare degli strumenti per manipolare basi di conoscenza (ci concentriamo su tesauri e ontologie) operando in modo tale da rendere compatibili informazioni tratte da fonti differenti.

Il testo [23] presenta una trattazione teorica approfondita sulle metodologie per sviluppare un'ontologia: in particolare il framework NeOn prevede vari scenari tipici in cui ci si può trovare quando si voglia costruire un'ontologia, fa particolare attenzione ai casi nei quali siano già presenti informazioni, ma queste vanno riorganizzate.

### Esempio

Consideriamo una semplice ontologia che rappresenta persone con legami di parentela genitore-figlio; le persone hanno uno o più nomi salvati nel tag `comment`. Modelliamo questa ontologia con una classe `Persone` e una sottoclasse `Genitori` (i cui individui sono `Persone` che realizzano la relazione `genitoreDi`). Creiamo la relazione `genitoreDi`. Infine popoliamo l'ontologia con alcuni individui. Il risultato ottenuto con Protégé è un documento XML di questo tipo:

Listato 1.1: persone.rdf

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.persone/"
     xml:base="http://www.persone/"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:www="http://www.persone#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://www.persone"/>
    <!-- http://www.persone#genitoreDi -->
    <owl:ObjectProperty rdf:about="http://www.persone#genitoreDi"/>
    <!-- http://www.persone#Genitori -->
    <owl:Class rdf:about="http://www.persone#Genitori">
        <owl:equivalentClass>
            <owl:Restriction>
                <owl:onProperty
                ↪  rdf:resource="http://www.persone#genitoreDi"/>
                <owl:someValuesFrom
                ↪  rdf:resource="http://www.persone#Persone"/>
            </owl:Restriction>
```

[1] Un'estensione del World Wide Web in cui le informazioni siano comprensibili a un automa [4].

```
20          </owl:equivalentClass>
21          <rdfs:subClassOf rdf:resource="http://www.persone#Persone"/>
22      </owl:Class>
23      <!-- http://www.persone#Persone -->
24      <owl:Class rdf:about="http://www.persone#Persone"/>
25      <!-- http://www.persone#Bruto -->
26      <owl:NamedIndividual rdf:about="http://www.persone#Bruto">
27          <rdf:type rdf:resource="http://www.persone#Persone"/>
28          <rdfs:comment>Bruto</rdfs:comment>
29      </owl:NamedIndividual>
30      <!-- http://www.persone#Cesare -->
31      <owl:NamedIndividual rdf:about="http://www.persone#Cesare">
32          <rdf:type rdf:resource="http://www.persone#Genitori"/>
33          <www:genitoreDi rdf:resource="http://www.persone#Bruto"/>
34          <rdfs:comment>Caio</rdfs:comment>
35          <rdfs:comment>Augusto</rdfs:comment>
36          <rdfs:comment>Giulio</rdfs:comment>
37          <rdfs:comment>Cesare</rdfs:comment>
38      </owl:NamedIndividual>
39  </rdf:RDF>
```

Per quanto non sia impossibile leggere la struttura e i dati dal listato precedente, un modo naturale per rappresentare le ontologie è sotto forma di grafi. Qui vediamo il grafo[2] che mostra la struttura dell'ontologia e possiamo apprezzare quanto sia semplice la sua struttura rispetto a quello che avremmo potuto immaginare dal listato 1.1. Nel grafo non sono rappresentati gli individui, possiamo comunque leggere quanti ve ne sono per ogni classe (in questo caso un individuo di tipo `Persone` e uno di tipo `Genitori`)

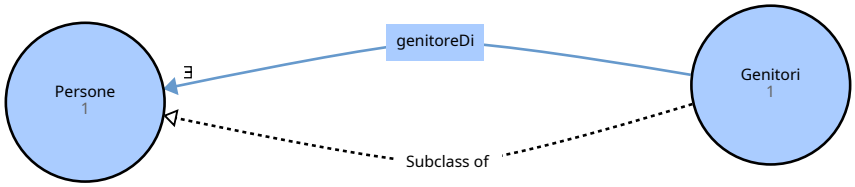


Figura 1.1: Grafo ontologia `persone`

### 1.2.2 Tesauri

#### Definizione

Nella loro accezione più generale possibile, i tesauri sono risorse nelle quali termini affini sono raggruppati assieme [13]. In particolare un tesauro fornisce un vocabolario preciso e controllato rispetto a un particolare dominio di interesse. Queste strutture possono aiutare il ricercatore a riformulare le strategie di ricerca fornendo una serie di sinonimi, contrari, definizioni e traduzioni in altre lingue del termine cercato [22].

Esistono diversi tipi di tesauro in base alle modalità di costruzione e fruizione [13]: nel nostro caso il tesauro sarà costituito da un vocabolario tassonomico in cui le relazioni tra oggetti sono di tipo BT (broader term), cioè ogni concetto può avere un riferimento a un concetto più generale, formando in questo modo una struttura ad albero.

[2]Tutti i grafi presenti un questo elaborato sono stati ottenuti grazie a http://vowl.visualdataweb.org/webvowl.html.

#### Espressività

Come si può immaginare, il potere espressivo di un tesauro è inferiore a quello di un'ontologia che non pone alcun limite alle relazioni definibili tra individui. In un tesauro inoltre ogni concetto ha, al più, un genitore, mentre in un'ontologia possiamo creare una classe che erediti le caratteristiche da più classi distinte.

## 1.3 Strumenti di editing

### 1.3.1 Protégé

Protégé[3] è uno strumento per la modellazione della conoscenza molto utilizzato; è un progetto open-source sviluppato all'università di Stanford e permette la manipolazione interattiva di ontologie e KB attraverso un'interfaccia grafica e delle API java.

Le funzionalità di Protégé possono essere aumentate grazie a componenti plug-in il cui numero è in continua crescita. Questi plug-in offrono nuovi metodi per la gestione delle ontologie, supporto per dati multimediali, engines per il ragionamento automatico e per l'interrogazione delle basi di conoscenza [21].

#### Descrizione del software

Protégé presenta un'interfaccia con numerose schede, ogni scheda permette all'utente di accedere a una differente funzionalità del software. Le schede base consentono di aggiungere nuova conoscenza e di effettuare ricerche nella KB.

Uno degli obbiettivi fondamentali di Protégé è quello di rendere l'inserimento e la ricerca in una base di conoscenza il più semplice possibile: mentre il sistema genera uno strumento per l'acquisizione della conoscenza, l'utente aggiunge informazioni riempiendo form intuitivi, selezionando elementi da liste e disegnando diagrammi.

Protégé permette di salvare le ontologie in numerosi formati, tra cui UML, XML, RDF e OWL.

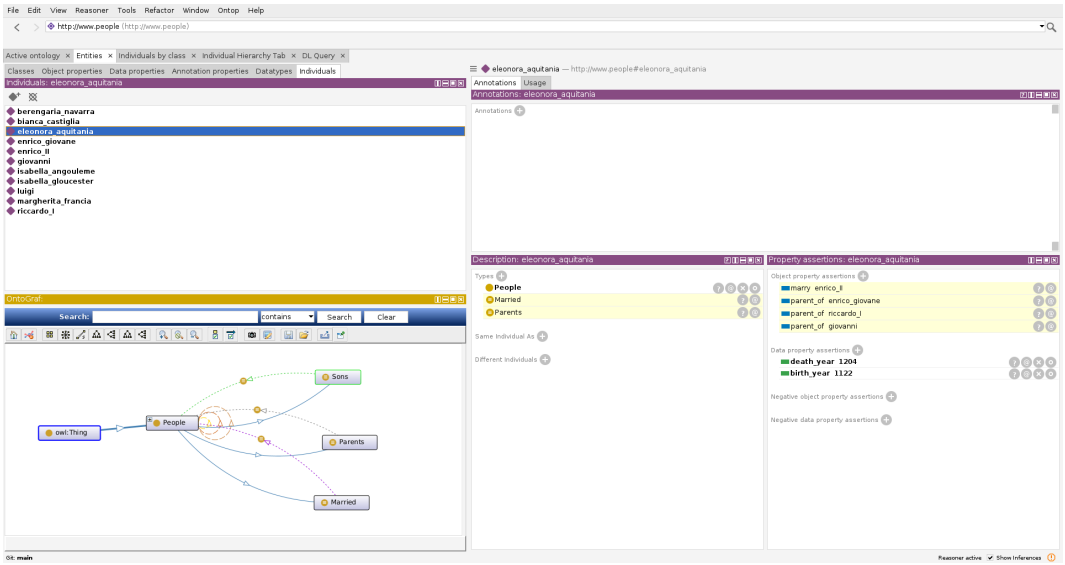


Figura 1.2: Interfaccia di Protégé

[3] https://protege.stanford.edu/

### 1.3.2 ℂDuce

ℂDuce è un linguaggio di programmazione funzionale, staticamente tipato e orientato allo sviluppo di applicazioni che lavorano su documenti XML [3]; nasce con l'intento di migliorare XDuce, un altro linguaggio di programmazione per il processing di file XML [10].

XDuce permette di manipolare unicamente documenti XML: questa sua specificità mostra come la presenza di feature mirate possano aiutare e semplificare lo sviluppo di applicazioni XML sacrificando, però, la possibilità di scrivere applicazioni complesse in cui non si facciano solo semplici trasformazioni di elementi XML.

ℂDuce si pone l'obbiettivo di integrare specifiche funzioni orientate alla manipolazione XML all'interno di un linguaggio potenzialmente general-porpouse. Per fare questo, ℂDuce usa un'algebra general-porpouse, con costruttori di tipo standard mantenendo l'espressività e la potenza delle espressioni regolari di XDuce attraverso l'uso di tipi ricorsivi e combinatori logici (unione, intersezione, differenza).

In ℂDuce è possibile creare complesse strutture dati, modellare il tipo di documenti XML, e interfacciarsi in modo relativamente semplice con altri linguaggi di programmazione [3].

### 1.3.3 Feature

Vediamo brevemente le principali feature di ℂDuce, non entreremo nel dettaglio essendo presente una guida dettagliata sul sito del progetto[4].

#### Verifica statica dei tipi

Il sistema di tipi è un componente centrale di ℂDuce; dal punto di vista pratico, un tipo è un insieme di valori identificati da un'espressione sintattica: questa descrizione permette di definire in modo naturale (come insiemi matematici) relazioni di sottotipo, connettivi logici nell'algebra dei tipi ed equivalenza di tipi.

I tipi ricoprono un ruolo importante anche nella parte statica del linguaggio: la correttezza di ogni trasformazione definita in ℂDuce è garantita staticamente. Inoltre in ℂDuce il pattern matching viene effettuato attraverso un'esatta inferenza di tipo: viene assegnata a ogni variabile legata il set esatto di tutti i valori che questa può assumere.

In questo modo è possibile costruire un sistema di tipi statico e molto preciso, che permette una migliore descrizione del comportamento dinamico del programma sviluppato.

#### Pattern matching

È un'operazione fondamentale in ℂDuce ed ha la forma:

```
match e with
  | p1 -> e1
  ...
  | pn -> en
```

Si cerca di fare il match tra la valutazione di un'espressione `e` e vari pattern `pi`. Il primo pattern che fa il match con `e` attiva la corrispondente espressione sulla destra che può usare le variabili legate dal pattern.

Il controllo statico dei tipi assicura che il pattern matching sia esaustivo, il tipo valutato per `e` deve essere sottotipo dell'unione dei tipi accettati dai pattern definiti sotto.

[4] https://www.cduce.org/

**Funzioni**

La forma generale di una funzione è:

```
fun myfunc (t1 -> s1; ...; tn -> sn)
  | p1 -> e1
  ...
  | pm -> em
```

In cui la prima riga è l'interfaccia della funzione, il resto è detto corpo:

- interfaccia: rappresenta il modo in cui la funzione si comporta: quando riceve un elemento di tipo `ti`, la funzione restituirà un elemento di tipo `si`; per ogni clausola `ti -> si` il sistema verifica che la funzione trasformi correttamente gli elementi;
- corpo della funzione: le funzioni operano per pattern matching degli argomenti: viene eseguita la prima trasformazione tale che l'argomento ricevuto come parametro faccia il match col pattern `pi`.

In ℂDuce è possibile definire anche applicazioni parziali (curried functions[5]) e funzioni di ordine superiore[6].

## 1.4 Metalinguaggi

Un metalinguaggio è, in generale, un linguaggio, o sistema di segni, naturale o artificiale, adottato per la descrizione della struttura formale di dati linguaggi[7]. In questo elaborato parliamo di linguaggi su tre differenti livelli:

- linguaggi di programmazione: sono i linguaggi con i quali sviluppiamo strumenti più o meno automatici per la manipolazione di basi di conoscenza;
- ontologie e tesauri: esprimono della conoscenza in modo formale, rispettano certe logiche per poter essere usate e restituiscono all'interrogatore una risposta che quest'ultimo sia in grado di comprendere;
- linguaggi per descrivere le basi di conoscenza: abbiamo bisogno di un linguaggio per descrivere la struttura di una KB, per definire quali relazioni possono esserci e in generale per parlare della KB stessa; chiameremo questi ultimi "metalinguaggi"[8] per distinguerli dai linguaggi di programmazione e dalla conoscenza che stiamo rappresentando.

Presentiamo alcuni metalinguaggi importanti che useremo per descrivere ontologie e tesauri.

### 1.4.1 XML

XML (Extensible Markup Language) è un formato testuale semplice e molto flessibile, inizialmente ideato per far fronte alla pubblicazione digitale su larga scala e diventato col tempo uno strumento importantissimo per lo scambio di moltissime informazioni sul Web [24].

[5] https://en.wikipedia.org/wiki/Currying
[6] https://en.wikipedia.org/wiki/Higher-order_function
[7] https://www.treccani.it/vocabolario/metalinguaggio
[8] Questi linguaggi non ci permettono di descrivere il dominio di interesse di per sé ma descrivono come descrivere il dominio stesso.

I documenti XML servono per memorizzare elementi detti entità che contengono dati formattati e non. I dati formattati sono costituiti da due componenti: il dato stesso e una parte di markup; quest'ultimo codifica la descrizione logica e la struttura di memorizzazione del documento. XML consente di imporre regole alla struttura logica e di memorizzazione del documento [7].

Per una descrizione dettagliata delle specifiche di XML si rimanda a [7], mentre [15] illustra nei particolari caratteristiche e funzionalità dello Schema XML. Ci limitiamo qui a definire la struttura di un elemento XML: `<(tag) (attr)>content</(tag)>` e a fornire un esempio di un'entità che potremmo trovare in un documento XML:

```
<person gender="F">
  <name>Clara</name>
  <children>
    <person gender="M">
      <name>Bob</name>
      <children>
      <children/>
    </person>
  </children>
  <email>clara@lri.fr</email>
  <tel>314-1592654</tel>
</person>
```

Questa entità descrive una donna di nome Clara, che ha un figlio maschio di nome Bob (senza figli), attribuiamo a Clara anche un numero di telefono e una e-mail.

Alla luce di questa descrizione si può provare a reinterpretare il listato 1.1 che descrive la relazione di parentela tra Giulio Cesare e Bruto.

### 1.4.2 RDF/RDFS

#### RDF

RDF (Resource Description Framework) è un framework per il processing di metadati[9]; un framework è un'architettura logica che supporta una prassi, una metodologia o un progetto[10], in questo caso definisce le regole per la descrizione della conoscenza. RDF permette l'interoperabilità tra applicazioni che si scambiano informazioni comprensibili a un automa.

Lo scopo di RDF è quello di automatizzare il processing di risorse Web; trova quindi applicazione in: ricerca di informazioni permettendo un motore di ricerca migliore, catalogazione della conoscenza, descrizione di una collezione di pagine che rappresentano un unico "documento" logico, ecc...[6]

RDF si basa sull'idea di rappresentare oggetti attraverso identificatori Web (URI[11]) e di descrivere le risorse in termini di semplici proprietà e valori. I valori possono essere tipi semplici (numeri, caratteri, stringhe) oppure altre risorse identificate con URI.

RDF permette quindi di esprimere semplici affermazioni riguardo risorse attraverso grafi i cui nodi rappresentano risorse e i cui archi rappresentano proprietà e valori. Per rappresentare qualsiasi informazione in un documento RDF si usa la sintassi XML (tag e namespace) [16].

[9] Esattamente come i metalinguaggi i metadati sono dati che descrivono altri dati.
[10] https://it.wikipedia.org/wiki/Framework
[11] Uniform Resource Identifiers.

#### RDFS

Uno Schema RDF (RDFS) definisce quali siano le proprietà valide che si possono esprimere in una particolare descrizione con RDF; permette inoltre di imporre caratteristiche o restrizioni alle proprietà e ai valori che queste possono assumere. Per identificare uno Schema RDF viene usato il meccanismo dei namespace XML[12].

Interpretare uno Schema RDF significa attribuire un valore semantico a ogni proprietà presente nella descrizione RDF, questo significa che, anche se un'applicazione non è in grado di capire lo schema, può comunque ricostruire il grafo di risorse descritto da RDF [18].

### 1.4.3 OWL

L'espressività di RDF e RDFS è limitata, potendo descrivere solo predicati binari, una struttura delle classi gerarchica e proprietà anch'esse gerarchiche con un dominio e un codominio.

Sono state individuate dal gruppo di ricerca "Web Ontology Working Group" una serie di casi in cui le ontologie devono poter essere descritte con linguaggi più espressivi rispetto a RDF e RDFS [1].

Presentiamo le funzionalità aggiunte da OWL Full: la versione più espressiva[13] del linguaggio OWL; per una descrizione più approfondita del linguaggio si rimanda al documento prodotto da W3C a riguardo[14].

#### OWL Full

OWL Full presenta la compatibilità massima coi documenti RDF/RDFS, ha la massima espressività, ma non è un sistema decidibile e completo. Descriviamo questo con il solo scopo di fornire l'elenco più completo possibile di funzionalità aggiunte per aumentare l'espressività di RDF/RDFS.

OWL Full[15] permette di:

- definire restrizioni sui domini delle relazioni in base alla classe (si può limitare, ad esempio, un individuo di classe mucca a mangiare solo piante, senza applicare la stessa restrizione ad altri animali);
- esprimere disgiunzione tra classi;
- definire classi tramite operazioni insiemistiche tra altre classi;
- definire restrizioni sulla cardinalità di una proprietà;
- definire caratteristiche particolari di una proprietà come transitività, riflessività, proprietà inversa, ecc...

In OWL Full è, inoltre, possibile utilizzare ogni primitiva del linguaggio OWL combinandola con le primitive del linguaggio RDF/RDFS in modo da alterarne il significato originario (applicando anche una primitiva all'altra).

[12] Una serie di nomi, identificati da un URI usati in un documento XML come tipi di un elemento e nomi di attributi [5].
[13] E anche più pesante computazionalmente
[14] https://www.w3.org/TR/2004/REC-owl-features-20040210
[15] Come anche OWL DL [1].

### 1.4.4 SKOS

SKOS (Simple Knowledge Organization System) è un modello basato su RDF che permette di esprimere la struttura base e il contenuto di schemi concettuali come tesauri, tassonomie e altri tipi di vocabolari affini [11].

In SKOS i concetti possono essere identificati con URI, etichettati con stringhe di caratteri in una o più lingue, descritti con delle annotazioni, associati ad altri concetti od organizzati in strutture gerarchiche informali [17].

Più avanti vedremo che questo linguaggio è stato usato per descrivere il tesauro "Eurpeana Fashion Thesaurus": un tesauro facilmente reperibile online, ottimo per i nostri esperimenti di editing (ne parleremo nel capitolo 3).

## 1.5 Conclusioni

Ora che abbiamo definito i concetti base che incontreremo nell'elaborato, possiamo passare alla parte più pratica in cui vedremo come manipolare i metalinguaggi attraverso programmi in ℂDuce (Capitolo 2), come la trasformazione di tag ci permetta di passare da una rappresentazione della conoscenza all'altra (capitolo 3) e come possiamo fare inferenza e ricavare informazioni da una base di conoscenza direttamente attraverso ℂDuce senza ricorrere a reasoner esterni (capitolo 4).

# Capitolo 2

# Strumenti offerti da ℂDuce

## 2.1 Introduzione

In questo capitolo analizziamo brevemente gli strumenti principali offerti da ℂDuce per l'editing di documenti XML. La trattazione vale per un qualsiasi documento XML, ma negli esempi ci concentreremo sulle ontologie analizzando i primi comandi per trattare ontologie dalla struttura semplice.

## 2.2 Parsing di documenti XML

Un documento XML non è altro che una struttura ad albero: ogni nodo rappresenta un concetto che può essere meglio definito nei figli del nodo stesso.

In ℂDuce possiamo ricostruire tale struttura definendo dei tipi. La forma generale di un elemento XML è `<(tag) (attr)> content` dove `tag`, `attr` e `content` sono espressioni su cui è possibile fare patten matching. Si può quindi creare un tipo generale che faccia match con il tag root del documento XML e che, come `content`, abbia un array eterogeneo che conterrà tutti i figli del tag root. A ogni elemento del vettore sarà associato un tipo che avrà la stessa struttura del tipo generale e che permetterà di descrivere la struttura del documento XML discendendo fino alle foglie.

### 2.2.1 Esempio

Torniamo all'esempio dell'ontologia di persone definita nel listato 1.1 e descriviamo in ℂDuce la sua struttura:

Listato 2.1: persone.cd

```
1 namespace owl="http://www.w3.org/2002/07/owl#"
2 namespace rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3 namespace www="http://www.persone#"
4 namespace xml="http://www.w3.org/XML/1998/namespace"
5 namespace xsd="http://www.w3.org/2001/XMLSchema#"
6 namespace rdfs="http://www.w3.org/2000/01/rdf-schema#";;
7
8 type Ontology = <rdf:RDF xml:base=String> [ Thing * ]
9 type Thing = Ont | Property | Class | Individual
```

```
10
11 type Ont = <owl:Ontology rdf:about=String> []
12
13 type Property = <owl:ObjectProperty rdf:about=String> []
14
15 type Class = <owl:Class rdf:about=String> [ ClassAttr * ]
16 type ClassAttr = EqClass | SubClass
17 type EqClass = <owl:equivalentClass> [ <owl:Restriction> [ AnyXml * ] ]
18 type SubClass = <rdfs:subClassOf rdf:resource=String> []
19
20 type Individual = <owl:NamedIndividual rdf:about=String> [ IndAttr * ]
21 type IndAttr = TypeInd | PropInd | Name
22 type Name = <rdfs:comment> String
23 type TypeInd = <rdf:type rdf:resource=String> []
24 type PropInd = <_ rdf:resource=String> [];;
25
26 let ontology :? Ontology = load_xml "persone.rdf";;
```

Si definiscono (linea 1 a 6) i namespace in modo che ℂDuce li possa interpretare correttamente nel resto del documento. Si passa poi alla struttura vera e propria del file: tutte le informazioni sono contenute nell'elemento di tipo `Ontology` che è costituito da un tag e da una lista di elementi di tipo `Thing` che a sua volta può rappresentare un elemento di tipo `Ont`, `Property`, `Class` o `Individual`. Ognuno di questi elementi fa il match con un nodo figlio del nodo root del documento XML; a loro volta questi elementi hanno una struttura interna che può essere più o meno dettagliata a seconda di come ci interessa operare sul documento. È interessante notare che, dato che in questo caso le specifiche proprietà della restrizione di un elemento `EqClass` non ci interessano, abbiamo potuto fare il match di queste con un vettore di lunghezza arbitraria (anche nulla) di generici elementi XML. Se avessimo voluto specificare che il tag restrizione contiene esattamente due elementi senza specificare quali, avremmo potuto scrivere `<owl>Restriction> [AnyXml AnyXml]`.

Quando andremo a caricare il documento con il comando alla riga 26, ℂDuce carica il file ed esegue il controllo di tipo verificando che la struttura del file XML sia effettivamente quella descritta dall'elemento `Ontology`; se il controllo va a buon fine, otterremo l'elemento chiamato `ontology` di tipo `Ontology` contenente tutto il file XML.

## 2.3 Manipolare documenti XML

Una volta definita la struttura del documento possiamo definire delle funzioni che mappano un elemento in un altro elemento. Gli strumenti fondamentali sono:

- pattern matching: descritto nel paragrafo 1.3.3
- `map`: permette di applicare una funzione a tutti gli elementi di una lista e restituisce una nuova lista di elementi trasformati della stessa lunghezza della lista iniziale;
- `transform`: permette di applicare una funzione a ogni elemento di una lista, restituendo per ogni elemento una lista di lunghezza arbitraria (anche nulla) e concatenando infine il risultato. Grazie alla funzione `transform` si può ottenere una lista di lunghezza diversa rispetto alla lista di partenza.

### 2.3.1 Esempio

Consideriamo nuovamente l'ontologia padri-figli con struttura definita nel listato 2.1 e costruiamo due funzioni, la prima (chiamata `name`) che permetta di estrarre da un individuo tutti i nomi, la seconda (chiamata `names`) che usi la prima per costruire una lista con tutti i nomi contenuti nell'ontologia.

Listato 2.2: basic functions

```
(*first function*)
let fun name (Individual -> [String *])
  <owl:NamedIndividual ..> [ (n::Name | _) * ] -> map n with
    <rdfs:comment> s -> s;;

(*second function*)
let fun names (Ontology -> [String *])
  <rdf:RDF ..> x -> transform x with
    | (x & Individual) -> name x
    | _ -> [];;
```

Nella prima funzione, usando il pattern matching, si lega la variabile `n` alla sequenza di tutti gli elementi di tipo `Name` associati a quell'individuo, poi si usa `map` per estrarre da ogni elemento di questa lista solo la stringa col nome.

Nella seconda funzione si usa `transform` per selezionare dalla lista `x` (lista di elementi tipo `Thing`) i soli elementi `Individual`, a questi si applica la funzione `name` sopra definita. La seconda clausola di `transform`, che serve a scartare tutti gli elementi di cui non si è ancora fatto il match, è implicita e può essere quindi omessa.

La prima funzione ci ha permesso di passare da una lista `n` di elementi `Name` a una lista della stessa lunghezza di stringhe. La seconda ci ha permesso di passare da una lista di elementi `Thing` a una lista di elementi `Individual` di differente lunghezza (questa lista viene poi trasformata in una lista di stringhe usando la funzione `name`).

## 2.4 Query

Un punto di forza di ℂDuce sono le Query. Le useremo profusamente nei capitoli 3 e 4 per creare liste e selezionare elementi in modo rapido e leggibile al posto della funzione `transform`. La forma generale di una Query è molto simile alla stessa espressa in SQL:

```
select e
from p1 in e1,
   p2 in e2,
   :
   pn in en
where c
```

Dove `e` ed `ei` sono espressioni, `pi` sono pattern e `c` è un'espressione booleana. Il risultato finale è la lista di tutti i valori ottenuti valutando `e` nella sequenza di possibili combinazioni in cui le variabili libere di `e` sono legate facendo il match dei pattern `pi` con le espressioni `ei`, a condizione che `c` sia rispettata.

Le Query possono essere simulate con `transform` che permette di creare, grazie a pattern matching successivi, l'espressione `e` e di selezionarla solo nel caso in cui l'espressione booleana `c` sia rispettata. Si può riscrivere la generica Query usando la `transform` come segue:

```
transform e1 with p1 ->
  transform e2 with p2 ->
    ...
      transform en with pn ->
        if c then  [e] else []
```

La valutazione dei due comandi produce il medesimo risultato.

Anche se `transform` può sembrare più versatile o espressiva delle Query, quando possibile, è sempre vantaggioso usare queste ultime. I vantaggi sono molteplici:

- la struttura delle Query è più elegante e leggibile, permette di capire con più facilità cosa si sta cercando;
- le Query sono automaticamente ottimizzate con le stesse tecniche di ottimizzazione di SQL, questo è molto vantaggioso soprattutto in ontologie o tesauri particolarmente popolati;
- CDuce fornisce dei controlli a priori sul risultato della Query permettendo di evidenziare errori che porterebbero a una Query che viene interpretata correttamente ma produce sempre risultati vuoti.

Oltre che con la struttura del "`select from`" le Query possono essere espresse anche come proiezioni, come mostrato nel listato 2.3.

### 2.4.1 Esempio

Facendo sempre riferimento all'ontologia con struttura definita nel listato 2.1, scriviamo delle Query per ottenere lo stesso risultato della seconda funzione definita nel listato 2.2:

Listato 2.3: basic Query

```
(*Query*)
let sel = select y
    from x in [ontology]/Individual,
         y in [x]/Name
in
map sel with <rdfs:comment> n -> n;;

(*projection 1*)
map [ontology]/Individual/Name with <rdfs:comment> n -> n;;

(*projection 2*)
map [ontology]/Individual/<rdfs:comment> _ with <rdfs:comment> n -> n;;

(*error 1*)
[ontology]/Name;;

(*error 2*)
let sel = select x
    from x in [ontology]/Individual
in [sel]/Name

```

```
22 (*correction error 2*)
23 let sel = select x
24     from x in [ontology]/Individual
25 in sel/Name
```

La Query crea una lista di elementi di tipo `Name` a cui si assegna il nome `sel`, questa lista viene poi trasformata in una lista di stringhe grazie alla `map`. La proiezione fa esattamente la stessa cosa in modo ancora più conciso. Sia nella forma del "`select from`", sia nella forma delle proiezioni, si possono usare anche espressioni per fare il match (oltre che tipi), come si vede nella `projection 2` (linea 11).

Gli ultimi due esempi (linee 14 e 17) mostrano come ℂDuce possa aiutare nella rilevazione degli errori: se eseguiti restituiscono rispettivamente gli avvertimenti: "`Warning: This projection always returns the empty sequence`" e "`Warning: This branch is not used`", informandoci che il risultato sarà sempre vuoto.

Analizziamo il primo errore: ℂDuce sa che, affinché il risultato possa contenere dei valori, gli elementi della lista chiamata `ontology` (che sono di tipo `Thing`) devono essere sottotipi di `[<_ ..>[Any* Name Any*] *]`; dato che così non è, il risultato sarà sempre vuoto.

Il secondo errore è dovuto al fatto che mettendo le parentesi quadre intorno a `sel` (linea 20) questo viene interpretato come una sequenza (anche se lo è già); ne risulta una sequenza il cui unico elemento è una sequenza (in particolare il tipo attribuito da ℂDuce è `[ [ Individual* ] ]`). Anche in questo, caso controllando il tipo, ℂDuce è in grado di capire che il risultato sarà sempre vuoto. Perché la Query restituisca dei valori, il tipo deve essere `[ Individual* ]`, cosa che avviene nella Query successiva (linea 22) che in effetti restituisce il risultato atteso.

## 2.5 Refactor in ℂDuce

Con i semplici strumenti presentati fin'ora siamo già in grado di ristrutturare un'ontologia. Possiamo manipolarne la struttura, modificare le relazioni di sottoclasse, creare nuove relazioni, selezionare gruppi di elementi dei quali modificare, tramite funzioni, le caratteristiche.

## 2.6 Conclusioni

Abbiamo visto come si può descrivere la struttura di un documento XML in ℂDuce, come si può trasformare un elemento di un tipo in elemento di un altro tipo e come creare liste selezionando quali elementi aggiungere; abbiamo infine fornito uno spunto su come sia possibile ristrutturare un'ontologia. Nei capitoli 3 e 4 applicheremo gli strumenti analizzati a esempi notevoli che mostrino in che modo e in che contesti ℂDuce possa essere usato per operare, modificandoli, tesauri e ontologie

# Capitolo 3

# Trasformare un tesauro in un'ontologia

## 3.1 Introduzione

Qui vediamo come sia possibile usare ℂDuce per creare ontologie a partire da una rappresentazione della conoscenza già formalizzata in altri modi. Per illustrare il processo di creazione faremo riferimento a un esempio facilmente replicabile: la trasformazione di un tesauro in un'ontologia (per una trattazione più astratta e formale sulla re-ingegnerizzazione di un tesauro si veda [14]). Commenteremo le scelte effettuate e le strategie adottate. L'esempio permette una discussione lineare e senza eccessivi giri di parole mantenendo comunque una buona generalità qualora si applicassero le stesse scelte e strategie a diversi contesti.

Dopo aver trasformato il tesauro in un'ontologia con ℂDuce, illustriamo brevemente quali altri strumenti avremmo potuto utilizzare, in particolare cercheremo di ottenere risultati analoghi con Protégé discutendo vantaggi e svantaggi di ciascun approccio.

## 3.2 Vantaggi del passaggio da tesauro a ontologia

Potremmo essere interessati a trasformare un tesauro espresso in SKOS in un'ontologia espressa in OWL per varie ragioni: prima fra tutte, un'ontologia è una rappresentazione più formale della conoscenza e OWL ha un potere espressivo maggiore di SKOS; potremo quindi effettuare Query più avanzate e utilizzare strumenti di inferenza più potenti.

Per presentare questo argomento prendiamo in considerazione il tesauro: "Europeana Fashion Thesaurus"[1] e tentiamo di trasformarlo in un'ontologia. Prima di addentrarci nella parte tecnica forniamo una breve introduzione al tesauro che editeremo.

### 3.2.1 Europeana Fashion Thesaurus: capturing imagination

L'"Europeana Fashion project"[2] è un progetto che si è posto come obbiettivo quello di organizzare sotto una struttura gerarchica tutto ciò che riguardasse la moda, comprendendo anche sinonimi e contrari. Il tesauro si può trovare all'indirizzo: http://thesaurus.europeanafashion.eu/. Il tesauro permette di accedere in modo logico, organizzato e strutturato a una vasta conoscenza. Data la struttura ad albero le, relazioni tra oggetti

[1] http://thesaurus.europeanafashion.eu/
[2] http://www.europeanafashion.eu/

sono chiare e l'aggiunta o la ricerca di informazioni può essere fatta in modo rapido ed efficiente.

### 3.2.2 Svantaggi del tesauro

Per la sua struttura, il tesauro permette una rappresentazione della conoscenza puramente tassonomica; questo in certi contesti può essere una limitazione, ad esempio: nonostante nel tesauro siano presenti sia capi d'abbigliamento che materiali, non c'è nessuna possibilità di mettere in relazione i due concetti in modo formale; inoltre, se avessimo degli strumenti per fare inferenza, non saremmo in grado di dedurre se una bandana è un accessorio per il capo, perché quest'ultima non si trova nel ramo degli accessori della testa.

Il passaggio a un'ontologia permetterebbe di costruire relazioni più complesse per descrivere in modo più ricco gli oggetti del dominio di interesse.

## 3.3 Struttura del tesauro

La struttura del tesauro che vogliamo re-ingegnerizzare è semplice: in particolare abbiamo una serie di concetti fondamentali da cui partono tutti gli altri rami che rappresentano i concetti derivati. I concetti fondamentali, almeno per quello che ci interessa, sono:

- oggetti di moda;
- colori;
- tecniche;
- materiali.

Ogni concetto, che sia o meno uno di quelli fondamentali, è rappresentato con un tag `Description`, contenente, a sua volta, una lista di attributi tra cui le `label` che contengono il nome (nelle varie lingue) del concetto, il tag `broader` che nei concetti derivati permette di risalire al nodo padre, le `scopeNote` che contendono una descrizione del concetto, e il tag `exactMatch` che rimanda a "The Getty Vocabularies"[3].

Analizzata la struttura del tesauro possiamo descriverla formalmente con ℂDuce in modo da fare il parsing del documento per poter definire funzioni che permettano di trasformare i concetti descritti dal tag `Description` in classi nel linguaggio OWL. Una possibile descrizione del tesauro in ℂDuce è la seguente:

Listato 3.1: thesaurus_europeana.cd

```
type Thesaurus = <rdf:RDF>[Desc *]
type Desc = <rdf:Description rdf:about=String>[DescAtt *]

type DescAtt = AltLabel | InScheme | PrefLabel | ScopeNote | ExactMatch |
↪  Broader | Type
type AltLabel = <skos:altLabel xml:lang=String> String
type InScheme = <skos:inScheme rdf:resource=String> []
type PrefLabel = <skos:prefLabel xml:lang=String> String
type ScopeNote = <skos:scopeNote xml:lang=String> String
type ExactMatch = <skos:exactMatch rdf:resource=String> []
```

[3] http://vocab.getty.edu/

```
type Broader = <skos:broader rdf:resource=String> []
type Type = <rdf:type rdf:resource=String> [];;
```

## 3.4 Struttura dell'ontologia

L'ontologia che vogliamo creare a partire da questo tesauro, almeno inizialmente, non potrà contenere più informazioni di quelle già presenti, ci limitiamo quindi a costruire una tassonomia, che potrà poi essere arricchita passando da un albero a un grafo, con relazioni più ricche tra le classi.

Una volta creata l'ossatura dell'ontologia, potremo andare a definire delle relazioni sugli individui ad esempio per specificare che un dato capo d'abbigliamento è prodotto con un determinato tessuto.

La struttura dell'ontologia, almeno per quello che ci serve per riportare tutte le informazioni contenute nel tesauro, può essere formalizzata in ℂDuce nel seguente modo:

Listato 3.2: ontology_europeana.cd

```
type Ontology = <rdf:RDF xml:base=String> [ Thing * ]
type Thing = Ont | Class

type Ont = <owl:Ontology rdf:about=String> []

type Class = <owl:Class rdf:about=String> [ ClassAtt * ]
type ClassAtt = SubClass | Label | Note | Dictionary
type SubClass = <rdfs:subClassOf rdf:resource=String> []
type Label = <rdfs:label xml:lang=String> String
type Note = <skos:scopeNote xml:lang=String> String
type Dictionary = <skos:exactMatch rdf:resource=String> []
```

Notiamo subito come la struttura dell'ontologia si sia semplificata rispetto alla struttura del listato 2.1, questo perché adesso ci interessa solo ricostruire i concetti SKOS con classi OWL.

È interessante come si possano integrare tag SKOS direttamente nel linguaggio OWL: in questo caso li usiamo per aggiungere informazioni umanamente leggibili (le note) e per mantenere il riferimento al dizionario (per una discussione dettagliata sull'interazione tra OWL e SKOS si veda [2]).

Si potrebbero anche eliminare del tutto i tag SKOS mappandoli adeguatamente in altri tag (ad esempio trasformando `scopeNote` in `comment`), ma questo non porterebbe nessun vantaggio dal punto di vista della formalità dell'ontologia e renderebbe più complesse le funzioni di trasformazione.

## 3.5 Da concetto SKOS a classe OWL

Il nostro obbiettivo è trasformare tutti i concetti del tesauro in classi di un'ontologia, mantenendo la gerarchia ed esprimendola come relazione di sottoclasse. Per raggiungere questo scopo definiamo una funzione per mappare gli elementi di tipo `DescAttr` in elementi di tipo `ClassAttr`. Successivamente usiamo queste funzioni per definirne una che ci permetta di passare da un intero concetto del tesauro a una classe dell'ontologia; infine, usando la

funzione `map` (Paragrafo 2.3), possiamo applicare questa funzione a tutti i concetti del tesauro per ottenere le classi che popoleranno l'ontologia.

### 3.5.1 Trasformare gli attributi

Iniziamo definendo le funzioni per mappare i tag del tesauro in tag dell'ontologia, questo è il primo esempio in cui si possono apprezzare i vantaggi dell'uso di un linguaggio funzionale. Esprimeremo le trasformazioni in modo semplice ed elegante senza perdere in leggibilità, inoltre avremo la garanzia che la trasformazione restituisca esattamente il tipo dichiarato nell'interfaccia della funzione, infine il controllo di tipo e l'inferenza del tipo di un'espressione ci aiuteranno a trovare e correggere eventuali errori. Definiamo una funzione per ciascun tag che desideriamo esportare, in particolare tralasciamo il tag `type` che nel tesauro assume solo due valori (`Concept` e `ConceptScheme`). Le funzioni possono essere scritte in ℂDuce come segue:

Listato 3.3: SKOS_to_OWL.cd

```
1  let fun transformNote (ScopeNote -> Note)
2    x -> x
3
4  let fun transformDictionary (ExactMatch -> Dictionary)
5    x -> x
6
7  let fun transformLabel (PrefLabel -> Label ; AltLabel -> Label)
8    | <skos:prefLabel xml:lang=l> lab -> <rdfs:label xml:lang=l> lab
9    | <skos:altLabel xml:lang=l> lab  -> <rdfs:label xml:lang=l> lab;;
10
11 let fun transformSubClass (Broader -> SubClass)
12   <skos:broader rdf:resource=res> [] -> <rdfs:subClassOf rdf:resource=res>
     ↪  [];;
```

Le prime due funzioni sono banali: confrontando i listati 3.1 e 3.2 notiamo che gli elementi di tipo `ScopeNote` e `Note` hanno la stessa struttura, così come gli elementi di tipo `ExactMatch` e `Dictionary`. Per questi tag è sufficiente la funzione identità.

Per quanto riguarda le `label`, nel tesauro ci sono due tipi di elementi, nell'ontologia abbiamo deciso di usarne solo uno. Vedremo quindi un esempio di overloading della funzione. Per quanto riguarda la trasformazione vera e propria, usiamo il pattern matching per legare la variabile `l` e `lab` rispettivamente alla lingua e al testo della `label` (linee 8 e 9); una volta legate queste variabili verranno usate per costruire il nuovo elemento di tipo `Label`.

La funzione più importante è certamente quella che lavora sull'elemento tipo `Broader`, questa si occupa infatti di trasformarlo in un elemento di tipo `SubClass` in modo da mantenere le relazioni gerarchiche del tesauro. Dal punto di vista della trasformazione però la funzione è molto semplice: alla riga 12 leghiamo un'unica variabile `res` alla stringa che identifica il nodo padre e con questa costruiamo il tag `subClassOf`.

### 3.5.2 Trasformare una singola classe

Per trasformare un singolo concetto espresso in SKOS in una classe OWL definiamo due funzioni: la prima trasforma un attributo del concetto (`DescAttr`) in un vettore di attributi di una classe (`[ ClassAttr * ]`) attraverso le funzioni definite prima (Listato 3.3). La ragione per cui il risultato deve essere un vettore, è che se l'attributo ci interessa, restituiamo

un vettore con un elemento, se l'attributo va scartato, restituiamo un vettore vuoto. La seconda funzione costruisce l'involucro esterno della classe e usa la prima per trasformare tutti gli attributi del concetto in attributi della classe. L'implementazione di queste funzioni potrebbe essere la seguente:

Listato 3.4: concept_to_class.cd

```
1  let fun transformAtt (DescAtt -> [ClassAtt*])
2    | x & PrefLabel  -> [(transformLabel x)]
3    | x & AltLabel   -> [(transformLabel x)]
4    | x & ScopeNote  -> [(transformNote x)]
5    | x & ExactMatch -> [(transformDictionary x)]
6    | x & Broader    -> [(transformSubClass x)]
7    | Any            -> [];;
8
9  let fun transformClass (Desc -> Class)
10   <rdf:Description rdf:about=ab> [ (descAtt :: DescAtt)* ] ->
11     let classAtt = flatten ( map descAtt with x -> transformAtt x) in
12     <owl:Class rdf:about=ab> classAtt;;
```

Le funzioni sono abbastanza semplici. Descriviamo brevemente la seconda: alla riga 10 usiamo il pattern matching per legare la variabile `ab` alla stringa che identifica il concetto (useremo questo identificatore anche per la classe); leghiamo poi la variabile `descAtt` al vettore di attributi del concetto SKOS, usando poi `map` per trasformare questo vettore in attributi di una classe OWL. Siccome la funzione `transformAttr` prende un elemento e restituisce un vettore, al termine della `map` avremo un vettore di vettori, per appiattirne la struttura usiamo la funzione già definita in ℂDuce `flatten`. A questo punto abbiamo tutti gli elementi per definire la nuova classe che costruiamo assemblando il tag `Class` con la stringa identificativa e il vettore di attributi.

## 3.6 Costruire la nuova ontologia

Ora che abbiamo una funzione per trasformare ogni concetto SKOS in una classe OWL, possiamo applicarla a tutti i concetti del tesauro per costruire un'ontologia. Possiamo completare la trasformazione in ℂDuce in questo modo:

Listato 3.5: thesaurus_to_ontontology.cd

```
1  let fashion :? Thesaurus = load_xml "thesaurus.rdf";;
2
3  let newClass = map [fashion]/Desc with x -> transformClass x
4  in
5  let newOnt : Ontology = <rdf:RDF
   ↪  xml:base="http://www.semanticweb.org/OntEur"> ( [ <owl:Ontology
   ↪  rdf:about="OntEur"> [ ] ] @ newClass )
6  in
7  dump_to_file_utf8 "Ontologia.rdf" (print_xml_utf8 newOnt);;
```

Alla linea 1, facendo riferimento alla struttura del tesauro definita nel listato 3.1, carichiamo il tesauro; alla riga 3 usiamo una proiezione per estrarre un vettore con tutti gli elementi di

tipo `Desc` (che sono i concetti del tesauro), usiamo la `map` per applicare a ognuno di questi elementi la funzione `transformClass` (Listato 3.4), infine diamo un nome al vettore di classi appena creato in modo da poterlo usare nel seguito della trasformazione.

Alla linea 5 creiamo l'ontologia vera e propria aggiungendo tutte le classi appena create, infine facciamo il dump su file generando un documento XML che potrà essere visualizzato con qualsiasi altro strumento incluso Protègè.

## 3.7 Versione compatta

Per trasformare il tesauro in un'ontologia, abbiamo definito varie funzioni che ci hanno permesso di trasformare pezzo per pezzo i tag SKOS nei rispettivi OWL. Assemblando progressivamente i pezzi, abbiamo costruito l'ontologia. Proviamo ora a sfruttare tutti i costrutti forniti da ℂDuce per riscrivere la trasformazione in una forma più compatta:

Listato 3.6: thesaurus_to_ontontology_compact.cd

```
1  let fun thesaurusToOntology (Thesaurus -> Ontology)
2    <rdf:RDF>[ (concepts :: Desc) * ] ->
3    let newClasses = map concepts with <rdf:Description rdf:about=ab> [
     ↪  (descAttr :: DescAtt)* ] ->
4      let classAttr = transform descAttr with
5        | x & ScopeNote -> [x]
6        | x & ExactMatch -> [x]
7        | <skos:prefLabel xml:lang=l> lab -> [<rdfs:label xml:lang=l> lab]
8        | <skos:altLabel xml:lang=l> lab  -> [<rdfs:label xml:lang=l> lab]
9        | <skos:broader rdf:resource=res> [] -> [<rdfs:subClassOf
         ↪  rdf:resource=res> []]
10     in
11     <owl:Class rdf:about=ab> classAttr
12   in
13   <rdf:RDF xml:base="http://www.semanticweb.org/OntEur"> ( [ <owl:Ontology
     ↪  rdf:about="OntEur"> [ ] ] @ newClasses );;
```

Questa funzione permette di passare direttamente da un elemento di tipo `Thesaurus` ad un elemento di tipo `Ontology`. Rispetto alla definizione più estesa abbiamo perso il controllo puntuale sul tipo degli attributi di tipo `ClassAttr`, infatti in questo caso ℂDuce ci assicura solo che siamo passati correttamente da attributi di un concetto ad attributi di una classe, senza controlli più specifici.

Nonostante questo minore controllo sui tipi, questa nuova definizione è molto più compatta rimanendo comunque leggibile: iniziamo con una `map` per trasformare il tag `Description` in un tag `Class`; dentro la `map` innestiamo una `transform` (linea 4) per trasformare gli attributi da SKOS a OWL per ogni elemento estratto con la funzione `map`. Siccome `transform` restituisce per ogni elemento una lista e infine le concatena, non abbiamo bisogno di applicare `flatten` al risultato che è già una lista di attributi di una classe.

## 3.8 Aumentare l'espressività

Ora che abbiamo ottenuto un'ontologia, possiamo aggiungere relazioni tra gli elementi, ad esempio possiamo mettere in relazione gli oggetti di moda con i materiali o i colori e le

tecniche con i materiali. Possiamo inoltre definire delle relazioni di sottoclasse più complesse di quella puramente tassonomica. In seguito (Paragrafo 4.2.1) presenteremo un'estensione della struttura dell'ontologia definita nel listato 3.2. Nell'estensione andremo ad arricchire la struttura con nuove informazioni e useremo la nuova versione per sviluppare selezioni e trasformazioni più sofisticate di quelle che avremo potuto definire su un tesauro. In figura si può vedere il risultato della trasformazione da tesauro a ontologia, l'immagine è troppo piccola per poter leggere le etichette, ma permette di apprezzare la struttura del tesauro, ogni albero rappresenta un concetto (trasformato in una classe) con le sue sottoclassi (per chiarezza sono state evidenziate le radici di ogni albero).

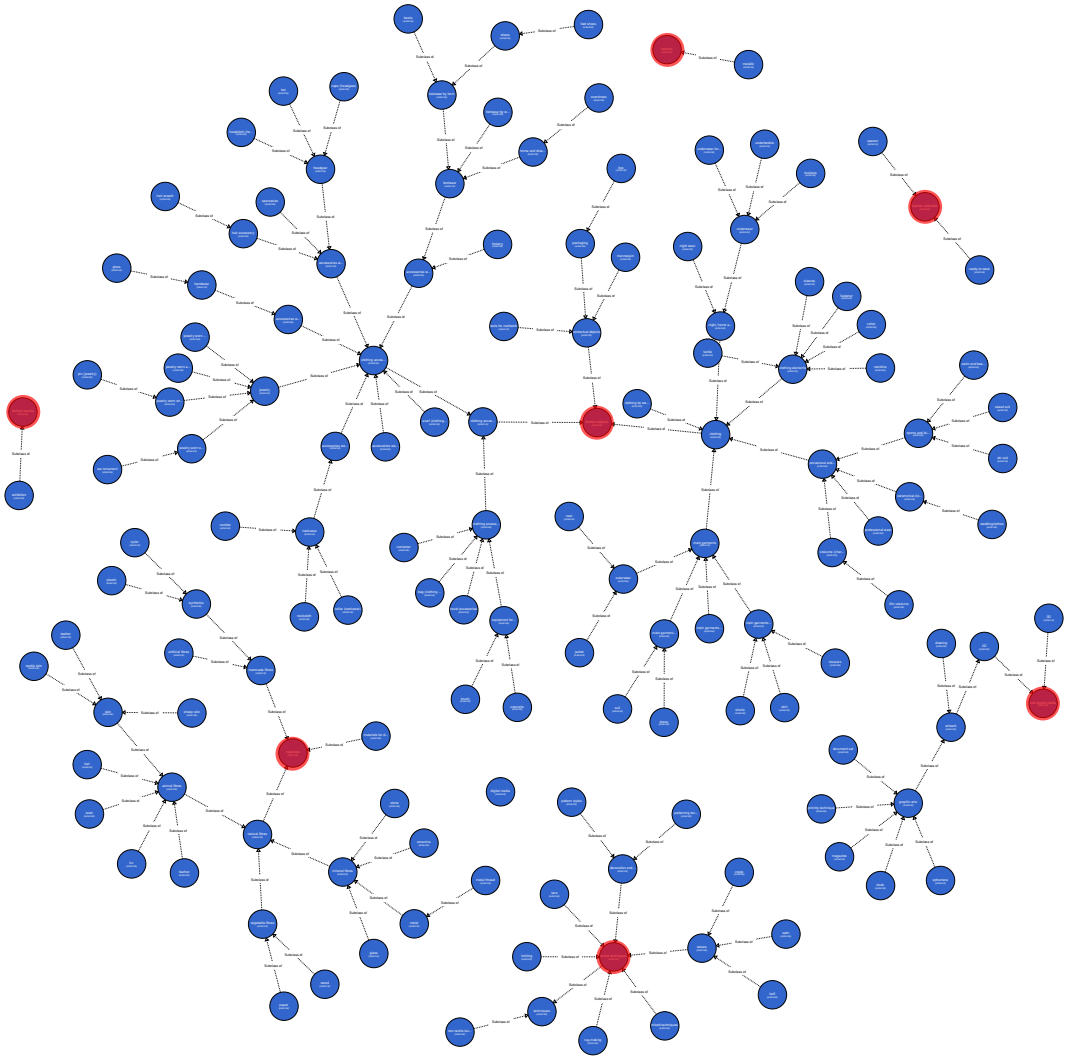

Figura 3.1: grafo ontologia `europeana`[4]

Nel capitolo 4 abbiamo riportato la struttura dell'ontologia con le nuove relazioni definite su essa (Figura 4.3).

[4]Il grafo non è completo, le sottoclassi sono troppe per ottenere un'immagine leggibile, WebVOWL permette di far collassare le classi operando una sorta di taglio in profondità del grafo.

## 3.9 In Protégé

Provando a editare il tesauro in Protégé, vediamo che vengono riconosciute come classi quella dei `Concepts` e quella dei `ConceptScheme`, tutti gli elementi del tesauro sono individui. In particolare i concetti fondamentali appartengono alla classe `ConceptScheme`, tutti gli altri alla classe `Concept`. Tra gli strumenti base di refactor di Protégé non ve ne sono per trasformare degli individui in classi, tanto meno mantenendo la struttura gerarchica specificata nel tesauro; non teniamo conto dei plug-in che permettono di scrivere programmi, ad esempio in java[5], per operare delle trasformazioni simili a quelle fatte con ℂDuce. Senza voler operare in modo puntuale su ogni individuo per trasformarlo manualmente in una classe, ciò che si può fare è comunque definire le stesse relazioni che abbiamo accennato precedentemente (Paragrafo 3.8), che avranno in questo caso dominio e range coincidenti (individui di classe `Concepts`). In questo modo possiamo comunque aumentare l'espressività del tesauro, ma con due grandi vincoli:

- non avremo il supporto dei reasoner per verificare la correttezza delle relazioni che valorizzeremo (ad esempio: siccome dominio e range della relazione "è fatto di" sono la stessa classe, potremo liberamente creare la relazione: "velluto è fatto di allumino");
- se volessimo definire delle relazioni di sottogenere più complesse, dovremo farlo tramite la definizione di relazioni, anziché con la più naturale definizione di sottoclasse.

## 3.10 Conclusioni

Nel contesto della re-ingegnerizzazione di un tesauro, ℂDuce si dimostra molto più versatile di Protégé, questo è dovuto principalmente a due fattori:

- come si legge in [14], una parte importante del processo di trasformazione è la conversione sintattica dei tag, in questo ℂDuce è uno strumento molto efficace;
- Protégé è uno strumento per editare ontologie espresse in OWL [19] e, nonostante sia molto potente e versatile, non è lo strumento più indicato per operare su un tesauro.

In questo frangente, l'uso di un linguaggio di programmazione che ci permettesse di definire esattamente come manipolare i dati è stato essenziale per ottenere un buon risultato di traduzione. Una volta acquisite delle competenze di base abbastanza solide con ℂDuce e avendo familiarità con gli strumenti che mette a disposizione, la scrittura di un programma che esegue la traduzione da tesauro a ontologia non risulta particolarmente complessa o lunga in termini di tempo (come testimonia la funzione del listato 3.6).

Dal punto di vista funzionale, abbiamo sfruttato la possibilità di definire funzioni avendo un sofisticato controllo sul tipo delle espressioni che stavamo trattando; questo ha permesso di evitare subito degli errori che, con linguaggi imperativi, non sarebbero emersi fino al momento dell'esecuzione. In particolare dovendo lavorare alternativamente su liste ed elementi singoli è capitato spesso che il tipo inferito fosse una lista, mentre il tipo richiesto fosse un elemento o viceversa (in un linguaggio come C, con l'uso di vettori e puntatori, un errore del genere avrebbe potenzialmente richiesto molto tempo per essere individuato e risolto).

[5] https://www.java.com/

# Capitolo 4

# Merge di ontologie

## 4.1 Intoduzione

In questo capitolo vediamo come sia possibile usare ℂDuce per fare il merge di ontologie, cercando di mettere in risalto quelli che possono essere i vantaggi di usare un linguaggio funzionale rispetto a uno strumento grafico. Alla fine del capitolo paragoneremo l'approccio con ℂDuce a quello con Protégé. Per mettere in evidenza i vantaggi di ℂDuce presentiamo un esempio un po' più complesso di quelli visti in precedenza in modo da far risaltare le potenzialità di ℂDuce e contemporaneamente dare un'idea di funzioni più complesse.

## 4.2 Obiettivo del merge

Fare il merge tra ontologie significa unire in modo consistente le informazioni contenute nelle varie ontologie di partenza, per ottenere un'ontologia finale in qualche modo più ricca e completa dei quelle di partenza. A seconda delle ontologie di partenza il merge può avere diversi scopi:

- ontologie che descrivono gli stessi concetti: lo scopo è unire ontologie che trattano dello stesso dominio, a cui potrebbero aver lavorato diversi gruppi di ricerca. Il merge deve mettere in evidenza i concetti comuni, i sinonimi e presentare tutte le informazioni raccolte dai diversi gruppi in modo consistente;
- traduzione di ontologie: avendo due o più ontologie in lingue diverse che trattano dello stesso dominio si vuole unire le ontologie in modo da creare corrispondenza tra concetti in lingue diverse, evidenziare le differenze espressive delle lingue e creare una base comune e condivisa del linguaggio;
- ontologie che trattano concetti diversi: si parte da ontologie che descrivono domini molto circostanziati per poi unirle in modo da rappresentare un modello più complesso e ricco di informazioni. Questo approccio permette di realizzare ontologie semplici limitando al massimo gli errori per poi unirle in modo da rappresentare un dominio costituito da molti concetti eterogenei. Ci troviamo in questo caso anche quando vorremmo poter usare ontologie già sviluppate come base di partenza per svilupparne di nuove.

In questo elaborato ci concentriamo sull'ultimo caso, illustrando come si possano unire 2 ontologie semplici e molto diverse fra loro per poter creare un modello di rappresentazione più ricco.

Le ontologie di cui faremo il merge in questo capitolo sono quelle che abbiamo già incontrato: partiamo dall'"European Fashion Thesaurus" e dall'ontologia per descrivere persone, per fonderle in una nuova ontologia che possa parlare degli usi e costumi della società, magari riferita a un ben preciso periodo storico. Vogliamo creare un'ontologia in cui sia possibile rappresentare delle persone, coi rispettivi legami di parentela e nella quale sia possibile associare alle persone uno status sociale, un lavoro e uno specifico abbigliamento.

### 4.2.1 Ontologie di partenza

Per creare la nuova ontologia consideriamo 3 ontologie di partenza:

- `society`: è lo scheletro dell'ontologia che vogliamo ottenere: è comodo crearla a priori in modo da non dover creare ex novo tutta la struttura in ℂDuce, inizialmente rappresenta delle persone sulle quali definiamo le relazioni `born_in`, `is` e `work_as` che associano una persona con la città di nascita, con il suo status sociale e con il proprio lavoro;

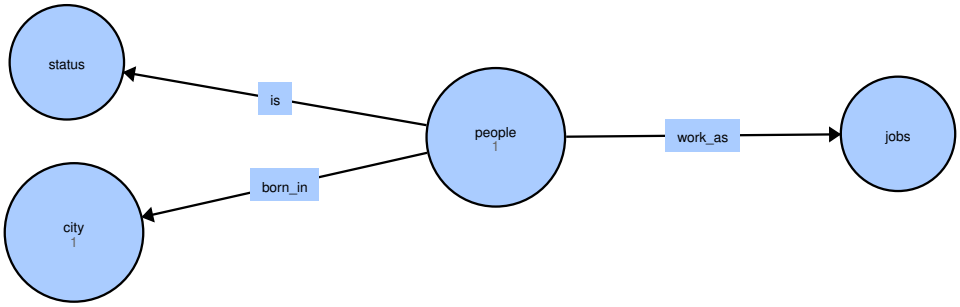


Figura 4.1: grafo ontologia `society`

- `people`: rappresenta gli individui con le loro parentele: rispetto all'ontologia presentata nel listato 1.1 abbiamo aggiunto la relazione simmetrica `marry`, e la relazione `son_of` come inversa di `parent_of`, aggiungiamo anche l'anno di nascita e morte.

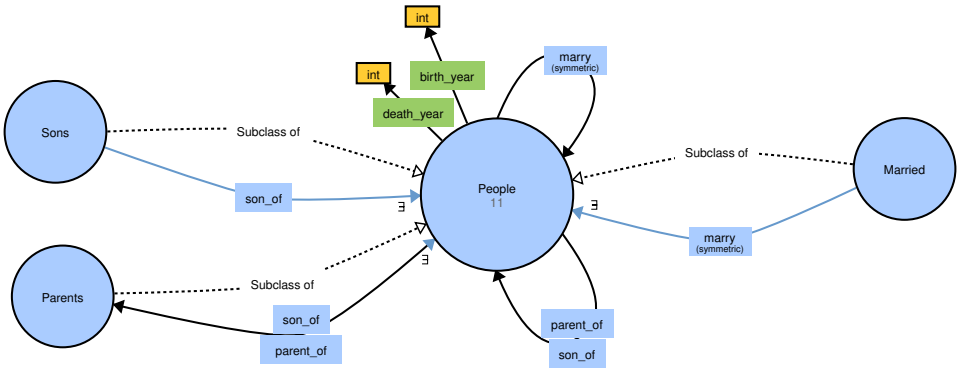


Figura 4.2: grafo ontologia `people`

- `fashion`: è l'ontologia che abbiamo creato nel capitolo 3 arricchita con le relazioni `made_of`, `crafted_with`, `color`, che indicano rispettivamente i materiali costituenti, le

tecniche realizzative e i colori di un capo d'abbigliamento. Per rappresentare l'ontologia in modo che abbia un'utilità si è deciso di operare un taglio in profondità e ampiezza del grafo, rappresentiamo per ogni classe principale solo il primo livello di sottoclassi, e nel caso questo livello fosse troppo numeroso viene tagliato in ampiezza.

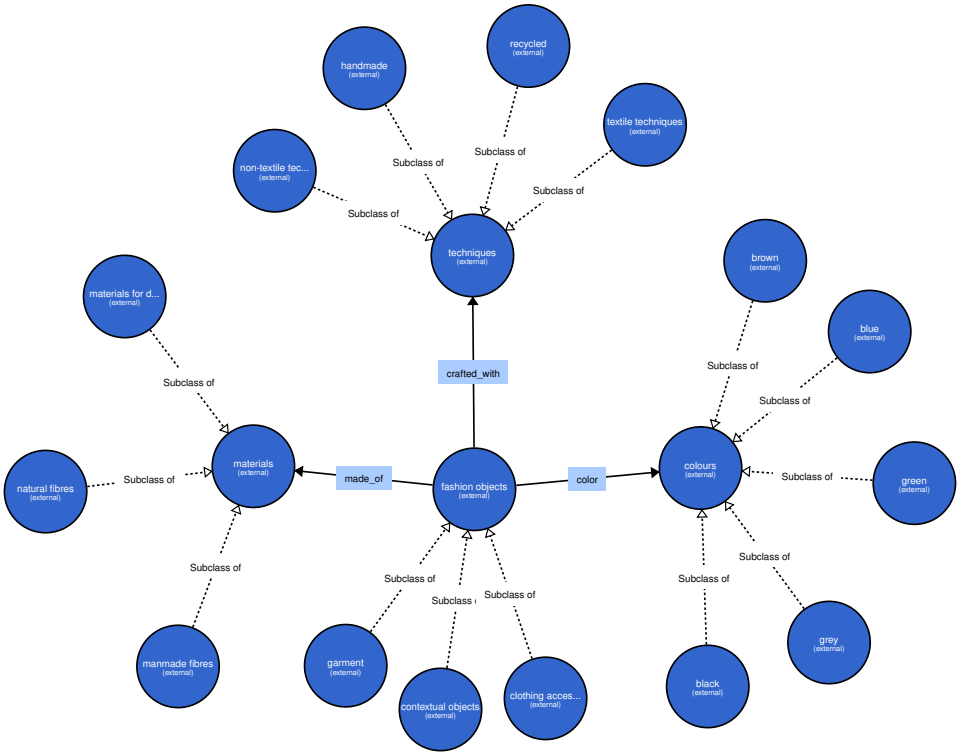


Figura 4.3: grafo ontologia `europeana`

### 4.2.2 Ontologia di arrivo

Vogliamo modificare `society` in modo da descrivere gli usi e i costumi della società del XII secolo, per fare questo estrapoliamo dall'ontologia `fashion` tutti i vestiti che non siano costituiti da fibre naturali e scremiamo dall'ontologia `people` tutte le persone che sono vissute in un'epoca che non ci interessa. Vogliamo poi che la classe `People` definita in `society` sia equivalente alla classe `People` definita in `people`, senza perdere tutte le relazioni di parentela, e potendo contemporaneamente sfruttare le nuove relazioni definite in `society`.

## 4.3 Struttura generale di un'ontologia

Per fare il merge dobbiamo poter caricare le tre ontologie in ℂDuce, quindi descriviamo una struttura generale per fare il parsing di un'ontologia generica:

Listato 4.1: general structure

```
type Ontology = <rdf:RDF xml:base=String> [ Thing * ]
type Thing = Ont | AnnProperty | ObjProperty | DataProperty | Class |
↪  Individual

type Ont = <owl:Ontology rdf:about=String> []

type AnnProperty = <owl:AnnotationProperty rdf:about=String> []

type DataProperty = <owl:DatatypeProperty rdf:about=String> [ PropAttr* ]
type ObjProperty = <owl:ObjectProperty rdf:about=String> [ PropAttr * ]

type PropAttr = Inverse | Domain | Range | PropType
type Inverse = <owl:inverseOf rdf:resource=String> []
type Domain = <rdfs:domain rdf:resource=String> []
type Range = <rdfs:range rdf:resource=String> []
type PropType = <rdf:type rdf:resource=String> []

type Class = <owl:Class rdf:about=String> [ ClassAtt * ]
type ClassAtt = SubClass | EqClass | Label | Note | Dictionary
type SubClass = <rdfs:subClassOf rdf:resource=String> []
type EqClass = <owl:equivalentClass> [ EqAttr ] | <owl:equivalentClass
↪  rdf:resource=String> []
type EqAttr = <owl:Restriction> [ AnyXml* ]
type Label = <rdfs:label xml:lang=String> String
type Note = <skos:scopeNote xml:lang=String> String
type Dictionary = <skos:exactMatch rdf:resource=String> []

type Individual = <owl:NamedIndividual rdf:about=String> [ IndAttr * ]
type IndAttr = IndClass | IndProp | DataProp
type IndClass = <rdf:type rdf:resource=String> []
type IndProp = <_ rdf:resource=String> []
type DataProp = <_ rdf:datatype=String> String
```

Rispetto alle strutture definite in precedenza notiamo varie aggiunte:

- si specifica meglio l'elemento `EqClass`: questo infatti può essere una restrizione (un genitore è una persona con dei figli) oppure un'equivalenza senza condizioni (ci serve per rendere uguali i concetti di persona definiti nelle due ontologie);
- definiamo il tipo `DataProperty`: questo rappresenta una proprietà degli individui, nel nostro caso la usiamo per specificare l'anno di nascita di una persona. Gli individui nella loro lista di attributi potranno ora averne uno di tipo `DataProp`;
- il tipo `AnnProperty` serve perché avendo editato l'ontologia `fashion` in Protégé (per aggiungere le relazioni non presenti nel tesauro) i tag SKOS sono stati correttamente riconosciuti [2] e classificati come tag `owl:AnnotationProperty`;

## 4.4 Merge

### 4.4.1 Funzioni utili

Prima di passare alla costruzione della nuova ontologia vediamo alcune funzioni utili che si possono applicare a qualsiasi ontologia e che sono servite per costruire le funzioni specifiche adatte a manipolare le particolari ontologie di interesse.

Listato 4.2: usefull function

```
let fun loadOntology (Latin1 -> Ontology)
  x -> load_xml x :? Ontology;;

let fun head ([ Any* ] -> Any)
  | ([ x ] @ _) -> x
  | [] -> "error empty list";;

let fun subClasses (Class -> Ontology -> [ Class* ])
  <owl:Class rdf:about=ab> [ _* ] -> fun (Ontology -> [ Class* ])
  ont ->
    select x from
           x in [ont]/Class,
           y in [x]/SubClass
    where (y = <rdfs:subClassOf rdf:resource=ab> []);;

let fun subClassesRec ( Class -> Ontology -> [ Class* ])
  cl -> fun (Ontology -> [ Class* ])
  ont ->
    let subCl = subClasses cl ont in
    subCl @ flatten (map subCl with y -> subClassesRec y ont)

let fun andList ( [ Bool* ] -> Bool )
  | ([ x ] @ y) -> (x && (andList y))
  | [] -> `true;;

let fun orList ( [ Bool* ] -> Bool )
  | ([ x ] @ y) -> (x || (orList y))
  | [] -> `false;;

let fun classOf (Individual -> Ontology -> [Class*])
  <owl:NamedIndividual ..> [ (tp::IndClass | _) *] -> fun (Ontology ->
  ↪  [Class*])
  ont ->
    transform tp with <rdf:type rdf:resource=str> [] ->
      transform [ont]/Class with x ->
        match x with <owl:Class rdf:about=a>[ _* ] -> if a = str then [x]
        ↪  else [];;

let fun contains (Any -> [Any*] -> Bool)
  obj -> fun ([Any*] -> Bool)
  lst ->
```

```
40    let intersect = select x from x in lst
41                    where (x = obj) in
42    match intersect with
43      | []       -> `false
44      | [ Any* ] -> `true;;
45
46  let fun isInClasses (Individual -> [Class*] -> Ontology -> Bool)
47    ind -> fun ([Class*] -> Ontology -> Bool)
48    classes -> fun (Ontology -> Bool)
49    ont ->
50    let cl = classOf ind ont in
51    let res = map cl with x -> contains x  classes in
52    andList res;;
```

Per la prima volta abbiamo fatto ricorso ad applicazioni parziali, questo è utile per poter parametrizzare anche l'ontologia di riferimento; dato che in questo caso ne manipoliamo contemporaneamente tre, è importante poter specificare volta per volta a quale ci riferiamo.

Vediamo per la prima volta degli esempi di funzioni ricorsive:

- `andList` e `orList` (linee 22 e 26) lavorano su liste di booleani e restituiscono il risultato della congiunzione o disgiunzione logica tra tutti gli elementi di una lista. Per ottenere questo risultato si fa il pattern matching della lista che può essere un elemento in testa alla lista seguito da una lista che chiamiamo coda; la funzione restituisce l'operazione logica tra la testa e la chiamata ricorsiva alla funzione, passando come parametro la coda della lista. Quando la lista è vuota (seconda clausola del matching) la funzione restituisce l'elemento neutro dell'operazione logica;

- `subClassRec` richiama ricorsivamente se stessa per costruire l'intero albero di sottoclassi a partire da una classe data (e dall'ontologia di riferimento). Ovviamente perché questa funzione possa terminare, la struttura delle classi deve essere un grafo aciclico (questa non è una grossa limitazione, infatti se la classe `c2` è contemporaneamente sopraclasse e sottoclasse di `c1`, allora `c1` e `c2` sono equivalenti e possono essere accorpate per eliminare i cicli, discorso analogo vale per cicli più lunghi).

Per verificare se un individuo si trova in un certo albero di classi, si potrebbe operare al contrario rispetto a `subClassRec`, risalendo l'albero fino a quando non si trova il padre desiderato (restituendo `true`) oppure la radice delle classi (restituendo `false`). Questo approccio, però, presenta alcune problematiche:

- un individuo può appartenere a più classi, quindi bisognerebbe risalire $n$ alberi dove $n$ è il numero di classi a cui appartiene l'oggetto;

- ogni classe può essere sottoclasse di più classi diramando così ulteriormente la ricerca.

Facendo alcuni test, si nota che un approccio di questo genere, oltre a essere impegnativo dal punto di vista implementativo, è poco efficiente dal punto di vista prestazionale. Come vedremo nel listato 4.3, per ogni capo d'abbigliamento dovremo chiederci se è costituito da materiali artificiali e questo rallenta il processo di merge. Per evitare questo problema costruiamo solo una volta la lista di materiali artificiali e, quando dobbiamo stabilire se un materiale è naturale o meno, verifichiamo se appartiene alla lista dei materiali artificiali. Questo approccio è vantaggioso in quanto la lista di materiali artificiali andrebbe creata in ogni caso per andare ad aggiungere alla nuova ontologia tutti i materiali che non lo sono

(conviene creare la lista dei materiali artificiali piuttosto che quella dei materiali naturali perché la prima contiene molti meno elementi, di conseguenza è più veloce da creare).

Per implementare questa ricerca usiamo `isInClasses` (linea 46) che prende un individuo e una lista di classi, valuta a che classi appartiene l'individuo (con `classOf`) e usa la funzione `contains` per creare una lista di valori booleani uno per ogni classe dell'individuo, verificando se è contenuto nella lista di classi fornita; infine si verifica se la lista di booleani contiene solo `true` con la funzione `andList`.

### 4.4.2 Selezione degli abiti

Mostriamo come si possano importare solo le fibre naturali e i vestiti costituiti esclusivamente da questi materiali, questo ha lo scopo di presentare come si possa fare della semplice inferenza sulla base di conoscenza usando solamente ℂDuce e senza ricorrere a reasoner esterni più sofisticati e complessi. Non mostriamo le funzioni per selezionare solo le persone vissute nel periodo storico considerato, questo per non inserire troppo codice e perché sarebbe poco istruttivo (non introdurremmo nuovi concetti rispetto a quelli che presenteremo nel seguito).

Per ragionare sul singolo abito ci domandiamo di che materiali è fatto usando la relazione `made_of` che associa uno o più materiali a un abito. Una volta che abbiamo i materiali, possiamo considerare la struttura ad albero dei materiali dell'ontologia `fashion` (la struttura è quella importata dal tesauro "European Fashion Thesaurus") per distinguere materiali naturali da artificiali (le due categorie devono essere disgiunte). Un vestito verrà considerato valido per la nuova ontologia solo se costituito unicamente da fibre naturali[1].

Una possibile implementazione delle funzioni che ci permettono di stabilire se un abito è costituito esclusivamente da fibre naturali è la seguente:

Listato 4.3: test artificial

```
1  let fun madeOf (Individual -> Ontology -> [Individual*])
2    <owl:NamedIndividual ..> [ (ip::IndProp | _) *] -> fun (Ontology ->
     ↪  [Individual*])
3    ont ->
4      transform ip with <ns1:made_of rdf:resource=str> [] ->
5        transform [ont]/Individual with x ->
6          match x with <owl:NamedIndividual rdf:about=a>[ _* ] -> if a = str
           ↪  then [x] else [];;
7
8
9  let fun isArtificial (Individual -> [Class*] -> [Class*] -> Ontology ->
   ↪  Bool)
10   ind -> fun ([Class*] -> [Class*] -> Ontology -> Bool)
11   materials -> fun ([Class*] -> Ontology -> Bool)
12   artificialMats -> fun (Ontology -> Bool)
13   ont -> if (isInClasses ind materials ont)
14     then (isInClasses ind artificialMats ont)
15     else
```

[1] Possiamo fare inferenza solo con i dati in nostro possesso, in particolare la classe dei materiali si divide in tre sottoclassi che rappresentano materiali naturali, artificiali e materiali di decorazione; su questi ultimi non possiamo fare alcun ragionamento sull'origine, si è operata una scelta permissiva considerandoli tutti naturali.

```
16      let matMade = madeOf ind ont in
17      orList (map matMade with x -> isInClasses x artificialMats ont);;
```

La funzione `madeOf` restituisce una lista di individui che sono tutti i materiali di cui è costituito un abito, per fare questo `madeOf` prende come parametri l'abito e l'ontologia di riferimento.

La funzione `isArtificial` ha due comportamenti differenti a seconda dell'individuo che le viene passato come parametro:

- materiale: se l'individuo appartiene all'albero dei materiali, il controllo consiste nello stabilire se questo materiale appartiene alla lista dei materiali artificiali: se è così si restituisce `true`;
- indumento: se l'individuo appartiene a una sottoclasse dei vestiti, prima si crea la lista di materiali di cui è costituito, poi si valuta: se almeno uno di questi è artificiale si restituisce `true`.

Per poter fare queste operazioni la funzione riceve come parametri: l'individuo da analizzare, la lista di tutti i materiali, la lista dei materiali artificiali e l'ontologia di riferimento.

### 4.4.3 Costruzione dell'ontologia

Assembliamo tutti i pezzi visti finora per manipolare l'ontologia `society` aggiungendo tutte le classi e gli individui che ci interessano. Presentiamo subito l'implementazione descrivendola poi passo passo:

Listato 4.4: assemble ontology

```
1  let fashion = loadOntology "europeana_formatted.rdf";;
2  let people = loadOntology "people.rdf";;
3  let society = loadOntology "society.rdf";;
4
5  let materials = subClassesRec <owl:Class
   ↪  rdf:about="http://thesaurus.europeanafashion.eu/thesaurus/10346"> []
   ↪  fashion;;
6  let artificialMats = subClassesRec <owl:Class
   ↪  rdf:about="http://thesaurus.europeanafashion.eu/thesaurus/10358"> []
   ↪  fashion;;
7  let fashionObjects = subClassesRec <owl:Class
   ↪  rdf:about="http://thesaurus.europeanafashion.eu/thesaurus/10000"> []
   ↪  fashion;;
8
9  let newMaterials : [Class*] = select x from x in materials
10                   where (not contains x artificialMats);;
11
12 let newFashionIndividual : [Individual*] = select x from x in
   ↪  [fashion]/Individual
13                           where (not isArtificial x materials
                             ↪  artificialMats fashion);;
14
15 let socPeople = head ([society]/<owl:Class
   ↪  rdf:about="http://www.semanticweb.org/society#people"> _) :? Class;;
```

```
16
17 let newSociety : [Thing*] = select x from x in [society]/(Thing \ Ont)
18                  where (not x = socPeople);;
19
20 let socPeople :? Class = match socPeople with <owl:Class rdf:about=str> [
   ↪  (attr::ClassAtt)* ] -> <owl:Class rdf:about=str>   (attr  @ [
   ↪  <owl:equivalentClass rdf:resource="http://www.people#People"> [] ]) ;;
21
22 let newPeople : [Thing*] = [people]/(Thing \ Ont);;
23
24 let newOnt :? Ont = <owl:Ontology
   ↪  rdf:about="http://www.semanticweb.org/society_merged"> [];;
25
26 let newThing : [Thing*] = [ newOnt ] @ newSociety @ [ socPeople ] @
   ↪  newFashionIndividual @  newMaterials @ newPeople;;
27
28 let newOntology :? Ontology = <rdf:RDF
   ↪  xml:base="http://www.semanticweb.org/society_merged">  newThing ;;
29
30 dump_to_file_utf8 "society_merged.rdf" (print_xml_utf8 newOntology);;
```

Da linea 1 a linea 3 carichiamo le ontologie che useremo, poi andiamo a creare le liste di classi di materiali che ci servono per utilizzare le funzioni definite nei listati 4.2 e 4.3: tutti i materiali e materiali artificiali. Creiamo anche la lista di tutte le classi di vestiti.

Alla riga 9 costruiamo la lista di classi dei materiali ammissibili per la nuova ontologia selezionando dalla lista totale tutti quelli non artificiali. In riga 12 estraiamo tutti i vestiti (individui) costituiti solo da materiali naturali (nella nuova ontologia inseriamo tutte le classi di vestiti, queste possono potenzialmente rappresentare individui fatti di fibre naturali).

Alla linea 17 prendiamo tutti gli elementi dell'ontologia `society` escluso l'elemento di tipo `Ont` e la classe `people`, questo perché l'elemento `Ont` verrà ricreato da zero alla linea 24 e la classe `people` andrà modificata per renderla equivalente alla classe `people` dell'ontologia `people` (linea 20)

In questo esempio prendiamo tutti gli elementi dell'ontologia `people`; come detto prima, presentare il codice per selezionare solo alcune persone sarebbe poco istruttivo. Una volta create tutte le nuove liste di elementi, le assembliamo alla linea 26 e, alla linea 28 costruiamo la nuova ontologia; in riga 30 la salviamo su file.

### 4.4.4 Risultato finale

L'ontologia che abbiamo costruito permette di descrivere la società nelle modalità che ci eravamo prefissati all'inizio del capitolo (Paragrafo 4.2), inoltre abbiamo importato già tutte le persone che avevamo modellato nell'ontologia `people` mantenendo le loro relazioni di parentela; possiamo definire nuove relazioni di parentela sulle persone già modellate in `society` (oppure attribuire loro una data di nascita) e usare le relazioni definite in `society` per arricchire la descrizione di un individuo presente in `people`. Abbiamo anche importato tutte le classi di vestiti e tutti i materiali naturali dall'ontologia `fashion`, possiamo ora creare nuove relazioni tra persone e vestiario per aggiungere informazioni sugli usi e costumi della società che intendiamo descrivere.

Del risultato finale riportiamo solo la parte di grafo che descrive le persone, la parte che descrive il vestiario è esattamente uguale a quella riportata nell'immagine 4.3 alla quale togliamo le fibre artificiali. Il grafo che descrive le persone ci fa apprezzare come

effettivamente le due classi estratte, una dall'ontologia `society` e l'altra dall'ontologia `people`, siano effettivamente state accorpate[2] in un'unica classe che possiede tutti gli attributi e le relazioni delle due classi di partenza.

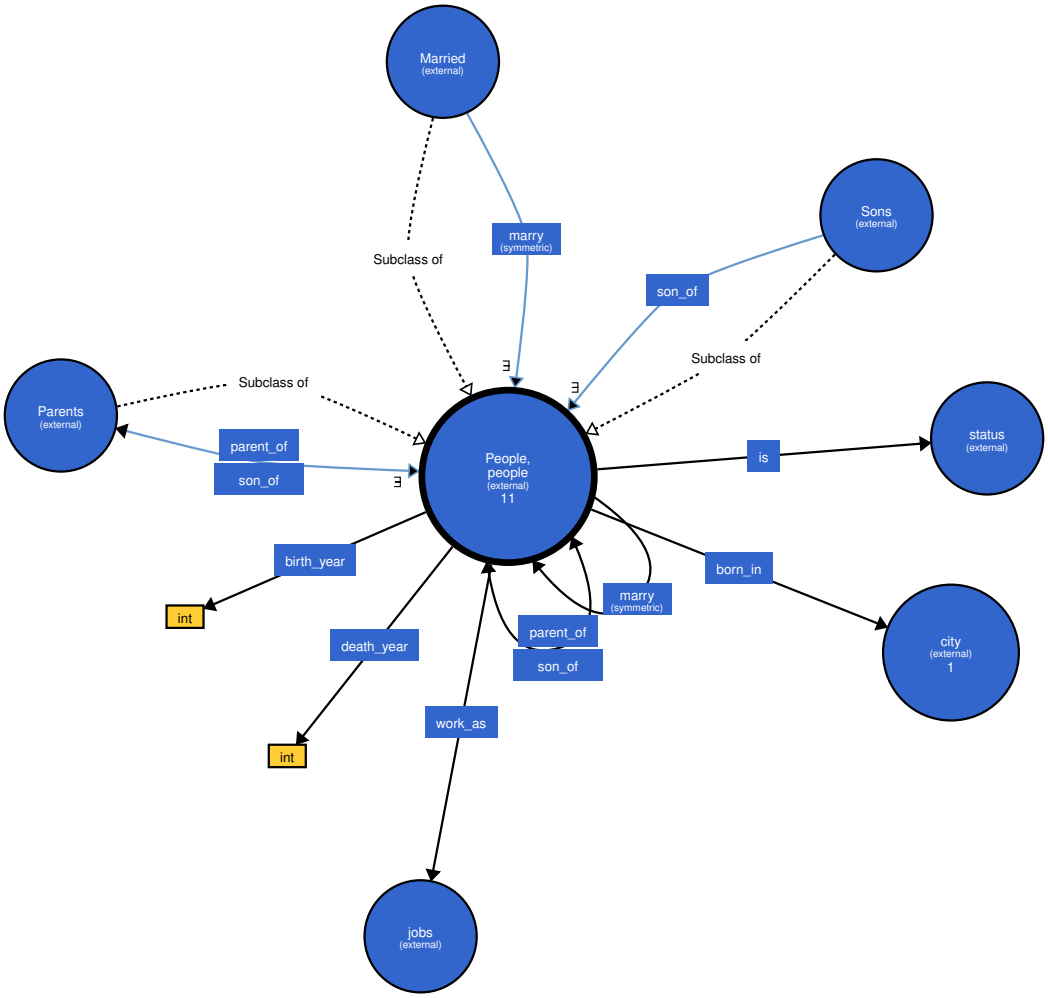


Figura 4.4: grafo ontologia `society merged`

## 4.5 In Protégé

In Protégé esiste un comando per fare il merge di ontologie: una volta aperte tutte le ontologie che siamo interessati a fondere, possiamo farne il merge e Protégé si occupa di creare una nuova ontologia e inserire tutte le classi, gli individui, le relazioni e le proprietà di tutte le ontologie di partenza.

Questo approccio è molto comodo se siamo interessati a prendere tutti i concetti dalle ontologie di partenza, in questo caso l'unico lavoro che ci rimane da fare è quello di mettere in relazione corretta le classi (in questo caso rendere equivalenti le due classi che descrivono le persone).

[2] In realtà rimangono due classi ma sono dichiarate equivalenti.

Nel caso in cui volessimo fare una selezione più fine su quali concetti importare dovremmo modificare le ontologie di partenza costruendo e scegliendo a prescindere (mediante delle Query) le classi e gli individui che vogliamo importare oppure, una volta fatto il merge, rimuovendo ciò che non ci interessa.

## 4.6 Conclusioni

Il merge con ℂDuce richiede di scrivere funzioni che permettano di selezionare cosa importare, questa precisione nella selezione comporta, però, più responsabilità da parte del programmatore che deve fare in modo che i dati continuino a essere consistenti e che le informazioni non vengano alterate durante il merge. Nuovamente sfruttiamo il controllo statico dei tipi offerto da ℂDuce: in questo caso possiamo notare che man mano che creiamo la nuova ontologia nel listato 4.4 verifichiamo che ogni elemento abbia effettivamente il tipo che ci aspettiamo. Controllando il tipo in fase di costruzione siamo sicuri che il documento che salviamo alla fine sia effettivamente un'ontologia e aprendola con un altro strumento (Protégé ad esempio) verrà riconosciuta correttamente e non ci saranno inconsistenze.

Nel caso specifico, il controllo di tipo è stato importante, infatti, la modifica della classe `socPeople` (Listato 4.4 riga 20) non era andata a buon fine e aveva generato un elemento XML che non faceva più il match con l'elemento di tipo `Class` (Listato 4.1). Avremmo comunque potuto salvare il risultato, ma se avessimo tentato di trattarlo come un file che descrive un'ontologia aprendolo con un altro software, avremmo ottenuto un messaggio d'errore (come se il file fosse corrotto) perché il programma non sarebbe stato in grado di interpretare il file come un file `.rdf` ben formato che descrive un'ontologia.

Il merge con Protégé, per quanto lineare, richiede comunque di formulare delle Query più o meno raffinate oppure di eliminare tutto ciò che non serve agli obbiettivi del merge, il processo quindi non è molto più guidato o automatico rispetto alla controparte in ℂDuce. Come si può vedere dal codice nei listati 4.2 e 4.3 la maggior parte del lavoro consiste nel creare funzioni che permettano di selezionare cosa importare, una volta fatto questo il codice del listato 4.4 che assembla l'ontologia risulta quasi banale. Selezionare elementi con funzioni in ℂDuce oppure con query in protégé richiede, in entrambi i casi, un certo impegno a livello di programmazione; considerando inoltre che in Protégé una volta selezionati gli elementi con delle Query è comunque necessario eliminare tutti gli altri, ℂDuce si rivela più efficace potendo importare esattamente ciò che si vuole (a patto di poterlo selezionare).

In conclusione Protégé è uno strumento vantaggioso nei casi in cui si voglia fare il merge tra ontologie mantenendo tutti i concetti rappresentati dalle ontologie di partenza, inoltre manipolando le informazioni con delle Query in Protégé siamo sicuri di ottenere delle ontologie consistenti. Quando è importante selezionare finemente i dati da importare, ℂDuce si rivela lo strumento migliore, avendo capacità espressiva e manipolativa superiore al set di strumenti base di Protégé (anche in questo caso non prendiamo in considerazione i plug-in che permettono di sviluppare programmi java per manipolare l'ontologia in Protégé). Con il controllo di tipi, ℂDuce fornisce comunque un certo supporto per verificare che le manipolazioni sulle ontologie portino a strutture consistenti.

# Capitolo 5

# Conclusioni

## 5.1 Maturità di ℂDuce

Prima di utilizzare un software in applicazioni importanti è bene domandarsi se lo stesso sia affidabile e venga manutenuto; questo al fine di evitare situazioni in cui tutto l'applicativo costruito al di sopra di uno strumento crolli al crollare delle fondamenta. Vediamo quindi qual è lo stato di sviluppo di ℂDuce e quali sono stati le difficoltà tecniche dell'utilizzarlo.

### 5.1.1 Il progetto ℂDuce

Come si può leggere nella pagina GitLab del progetto[1], ℂDuce nasce e si sviluppa come una ricerca condivisa tra il gruppo di ricerca sui linguaggi all'ENS[2] di Parigi e il gruppo di ricerca sui Database all'LRI[3] ad Orsay. Attualmente il progetto è mantenuto dai PPS laboratory[4] e dal Toccata Group[5].

Lo scopo del progetto è quello di costruire un linguaggio per manipolare documenti XML che usi seriamente i tipi XML; questo porta i vantaggi descritti nel paragrafo 1.3.3. L'implementazione attuale ha lo scopo di mostrare proprio queste feature innovative e di validare le scelte di progettazione effettuate.

Oltre a ispirarsi a XDuce[6], ℂDuce prende parte della sua sintassi da OCaml[7]. Le affinità tra OCaml e ℂDuce non si limitano alla sintassi, infatti per essere compilato ed eseguito ℂDuce richiede un'installazione funzionante di OCaml allineata alla versione di ℂDuce che si desidera utilizzare; su questa dipendenza torneremo dopo nel paragrafo 5.1.3.

### 5.1.2 Stato di sviluppo attuale

Sulla pagina del progetto[8] le ultime informazioni risalgono al 2021 e spiegano che gli sviluppatori si stanno impegnando in una riscrittura completa del compilatore per separare la libreria che gestisce i tipi dal resto del compilatore. La riscrittura del compilatore potrebbe provocare inconsistenza con le ultime versioni di OCaml pertanto si descrive un modo per aggirare il problema oppure si consiglia di utilizzare la vecchia versione. Nonostante le ultime

[1] https://gitlab.math.univ-paris-diderot.fr/cduce/cduce
[2] https://www.ens.psl.eu/
[3] https://www.lri.fr/
[4] https://www.irif.fr/
[5] https://toccata.gitlabpages.inria.fr/toccata/
[6] Come già discusso nel paragrafo 1.3.2.
[7] Un noto e molto utilizzato linguaggio di programmazione funzionale (https://ocaml.org).
[8] https://www.cduce.org/

informazioni sul sito siano del 2021 controllando la pagina GitLab si vede che, anche se il branch "stable" non riceve aggiornamenti da un anno, altri branch sono tutt'ora attivi[9].

### 5.1.3 Difficoltà incontrate

Ho testato ℂDuce su una distribuzione Ubuntu[10] e su Arch[11]; su Ubuntu non ho avuto successo (seguono dettagli), quindi tutte le prove descritte sono state fatte su Arch.

#### Ottenere ℂDuce

Sulla pagina del progetto si legge che ℂDuce è pacchettizzato per le più importanti distribuzioni Linux, ma questi pacchetti, almeno per Ubuntu risalgono a versioni molto vecchie (Ubuntu 12.10 e 13.04) e non funzionano nelle distribuzioni attuali.

Seguendo le istruzioni per compilare la nuova versione (sia polimorfa che monomorfa) di ℂDuce il processo fallisce, probabilmente questo è dovuto al fatto che da quando è stata scritta la guida a oggi, i pacchetti di OCaml sono stati ulteriormente aggiornati e il workaround descritto non è più sufficiente. Ho provato, questa volta con successo, a installare la vecchia versione di ℂDuce (version 0.6.1-rc1), ho scelto la versione monomorfa perché la versione polimorfa veniva descritta come "buggy and experimental".

Il processo di installazione descritto sul sito consiste a grandi linee nel:

- creare un ambiente virtuale per OCaml;
- installare in questo ambiente le vecchie versioni dei pacchetti necessari a compilare ℂDuce;
- compilare i sorgenti;
- ottenere l'eseguibile di ℂDuce.

#### Reperire informazioni

Sul sito si possono trovare molte risorse utilissime per imparare a usare ℂDuce, sono presenti in particolare due documenti:

- una guida utente[12] che spiega in modo molto rigoroso l'approccio del linguaggio ai vari tipi, operazioni e funzioni. La spiegazione è molto tecnica, presenta ogni argomento nel modo più generale possibile facendo uso estensivo dei pattern, per leggerla è necessaria qualche base sui linguaggi funzionali e su come il pattern matching operi (soprattutto se si intendono consultare solo alcune sezioni);
- un tutorial[13] che parte direttamente da alcuni esempi per spiegare il modo in cui ℂDuce opera. Gli esempi presentati sono molto ben curati e commentati, e permettono di cominciare subito a sperimentare con lo strumento.

Entrambe le guide hanno una controparte in PDF molto comoda per poter cercare all'interno dell'intero manuale un certo termine.

Le difficoltà sorgono nel momento in cui siamo interessati a certe sezioni del tutorial, alcune parti mancano per intero, ed essendo uno strumento relativamente di nicchia non c'è un forum o una community estesa alla quale si possa ricorrere per domande o dubbi. Questo

[9] Alla data in cui si scrive (10/10/2023) l'ultima attività risulta del 09/10/203.
[10] https://ubuntu.com/desktop
[11] I use Arch btw: https://archlinux.org/.
[12] https://www.cduce.org/manual.html
[13] https://www.cduce.org/tutorial.html

rende particolarmente difficile capire certe parti della guida utente che senza esempi risultano particolarmente oscure. In particolare durante gli esperimenti eseguiti è stato importante fare pattern sulle sequenze, questa sezione (come le precedenti di quel capitolo) è assente nel tutorial. Avendo delle basi di Haskell[14] e cercando di interpretare la guida che è piuttosto criptica in quel capitolo si può provare a immaginare quale sia la struttura corretta per fare il match (è servito nelle funzioni `andList` e `orList` nel listato 4.2).

### Output

Nei linguaggi funzionali le operazioni di input e output sono sempre delicate, in ℂDuce per poter esportare un documento XML dobbiamo fare due passi: prima trasformiamo l'elemento di tipo `AnyXml` (che contiene il file XML che vogliamo salvare) in una stringa poi facciamo il dump della stringa su file. Per la conversione da XML a stringa abbiamo due possibilità, se il documento contiene solo caratteri previsti dalla norma ISO-8859-1[15] si può usare `print_xml` altrimenti, per preservare tutti i caratteri, si usa `print_xml_utf8`. Ottenuta la stringa si può fare il dump su file con `dump_to_file` oppure `dump_to_file_utf8`, questo sarà un file XML che nel nostro caso rappresenta un'ontologia.

Provando ad aprire il risultato di un dump con un editor di testo ci si rende conto che tutto il codice XML si trova su una sola riga, manca completamente qualsiasi formattazione e il file risulta dunque illeggibile. Aprendo il file con uno strumento come Protégé non ci sono problemi, il file è ben formato e viene interpretato correttamente, ma se vogliamo vedere il risultato della manipolazione con ℂDuce senza ricorrere ad altri software appositi ci troviamo in difficoltà[16]. Se siamo intenzionati a vedere il risultato del dump di ℂDuce dobbiamo riformattare il documento per renderlo fruibile. Per fare questo ci sono molte opzioni, in questo caso è stato sviluppato un piccolo programma in java che riformatta il file XML rendendo possibile un'ispezione dello stesso mediante un editor di testo.

Il problema è probabilmente dovuto al fatto che le funzioni di `pretty-printing` di ℂDuce usano per le funzioni di OCaml, avendo forzato l'allineamento tra i pacchetti di OCaml e ℂDuce è possibile che non sia tutto esattamente compatibile, questo fa sì che le componenti non funzionali di ℂDuce e in generale l'interazione tra ℂDuce e OCaml non funzioni correttamente[17].

È in ogni caso un peccato che attualmente l'unico modo per poter leggere il risultato dell'esecuzione di un programma scritto con un linguaggio funzionale, usato senza ricorrere a nessuno stratagemma imperativo, richieda l'esecuzione di un programma in java per rendere leggibile il risultato.

### Dipendenza da OCaml

ℂDuce dipende da OCaml, questa dipendenza genera difficoltà dovute al fatto che le ultime versioni di OCaml non sono più allineate con ℂDuce, abbiamo già illustrato come si sia complicato il processo di installazione (Paragrafo 5.1.3), vediamo ora come il disallineamento tra ℂDuce e OCaml possa inficiare su delle feature del linguaggio.

Secondo quanto riportato sul sito del progetto[18] dovrebbe essere possibile richiamare delle funzioni di OCaml all'interno di ℂDuce e viceversa. La possibilità di unire OCaml e ℂDuce risulta particolarmente interessante permettendo di:

[14] https://www.haskell.org/

[15] https://www.iso.org/standard/28245.html

[16] Senza contare che Protégé quando si salva il file sul quale si ha lavorato sì lo riformatta correttamente, ma riposiziona gli elementi e aggiunge parti non presenti in origine (anche semplicemente i commenti) rendendo impossibile conoscere il documento originale.

[17] Torniamo sull'integrazione tra i due linguaggi nel successivamente (Paragrafo 5.1.3).

[18] https://www.cduce.org/manual_interfacewithocaml.html

- usare librerie di OCaml già esistenti per gestire database, network, interfacce grafiche, strutture dati, ecc...;
- usare ℂDuce come layer per gestire documenti XML all'interno di un progetto in OCaml;
- sviluppare codici in OCaml e ℂDuce perfettamente cooperanti che possando essere semplicemente assemblati nel progetto finale.

Le istruzioni per poter integrare OCaml e ℂDuce sono riportate nel file "`INSTALL`" scaricabile assieme ai sorgenti per compilare ℂDuce. Le istruzioni prevedono, oltre alla procedura descritta in precedenza, di avere i sorgenti di OCaml allineati alla versione di ℂDuce e di eseguire uno script di configurazione in cui si specifica il path dei sorgenti di OCaml.

Dalle prove effettuate il processo non sembra avere esito positivo, non viene indicato un workaround per appianare il disallineamento tra OCaml e ℂDuce[19] e dunque non è stato possibile testare questa interessante funzionalità del linguaggio.

Cercando una soluzione per integrare i due linguaggi si trova un progetto parallelo a ℂDuce ovvero OCamlDuce[20] che si propone di essere direttamente l'unione di OCaml e ℂDuce. Questo progetto purtroppo presenta gli stessi problemi di prima, per poter compilare il software serve una versione di OCaml, e dei suoi sorgenti, allineata a OCamlDuce[21].

Per quanto interessante la possibilità di integrare OCaml e ℂDuce presenta quindi notevoli difficoltà, inoltre anche se si riuscisse ad allineare perfettamente la versione di OCaml a quella di ℂDuce o di OCamlDuce, ne risulterebbe uno strumento poco pratico: lo scopo dell'integrazione è principalmente quello di poter sfruttare tutte le librerie e gli strumenti già esistenti sviluppati in OCaml, ma si dovrebbero usare sempre versioni obsolete di queste librerie e strumenti in modo da mantenere la compatibilità, questo ovviamente riduce moltissimo i benefici dell'integrazione.

### 5.1.4 Sviluppi futuri

Considerando che il progetto sembra essere ancora di interesse per i gruppi di ricerca che vi hanno lavorato e che il repository risulta attivo, e utilizzato, è auspicabile che la riscrittura del compilatore iniziata nel 2021 termini e la nuova versione di ℂDuce sia polimorfa e sfrutti l'ultima versione di OCaml, questo ridurrebbe drasticamente i problemi descritti sopra e potrebbe spingere più persone a interessarsi del progetto e a collaborare per rendere ℂDuce uno strumento professionale a tutti gli effetti; attualmente ℂDuce viene considerato dagli sviluppatori stessi come un prototipo di ricerca e non adatto ad applicazioni stabili[22].

## 5.2 Punti di forza di ℂDuce

Nonostante le difficoltà incontrate ℂDuce si è rivelato uno strumento molto potente e versatile per la manipolazione di ontologie, in particolare il fatto di essere un linguaggio funzionale lo rende particolarmente sintetico e leggibile una volta presa dimestichezza con i concetti base.

[19] Workaround descritto e necessario per l'installazione di ℂDuce stesso.
[20] https://www.cduce.org/ocaml.html
[21] l'ultima versione di OCamlDuce richiede OCaml 3.12.1, versione uscita nel 2011. La versione attuale di OCaml è la 5.1 .
[22] https://gitlab.math.univ-paris-diderot.fr/cduce/cduce/-/wikis/home

### 5.2.1 Sistema di tipi

I tipi descrivono un insieme di valori costruiti in un certo modo, è estremamente facile definire un tipo e la sua struttura, d'altra parte avere dei tipi definiti correttamente ci permette di scrivere funzioni che li manipolano come ci aspettiamo e che restituiscono un elemento esattamente del tipo che vogliamo.

Tutta la verifica dei tipi in ℂDuce avviene staticamente, saremo quindi sicuri che una funzione trasformi un concetto di un tesauro in una classe di un'ontologia ancora prima di eseguire questa funzione perché ℂDuce verifica che tutte le trasformazioni che applichiamo permettano di passare da un elemento del primo tipo a un elemento del secondo.

Il controllo di tipo avviene per tutte le funzioni che definiamo (a patto di specificarne correttamente l'interfaccia) e permette di scrivere del codice in cui la maggior parte del debugging possa essere fatta in fase compilazione e non a run-time.

### 5.2.2 Funzioni di ordine superiore

Come tutti i linguaggi funzionali anche ℂDuce permette di definire funzioni di ordine superiore, queste hanno numerosi vantaggi:

- possiamo scrivere funzioni semplici di cui è facile verificare la correttezza e assemblarle in funzioni più articolate che diventano più trattabili, gestendo in questo modo la complessità;
- le funzioni di ordine superiore permettono di generalizzare il codice in modo estremamente efficace, consideriamo il caso di una funzione che rimuove alcuni elementi di una lista secondo un certo criterio, nel momento in cui cambia il criterio dovremmo riscrivere almeno parte della funzione. Se invece implementiamo una funzione per la rimozione selettiva che fra i vari parametri accetti una funzione che specifica se un elemento della lista va eliminato, possiamo scrivere una funzione assolutamente generale che possa lavorare con un qualsiasi criterio che andremo volta per volta a specificare. Su questa possibilità torneremo dopo nel paragrafo 5.4.2

### 5.2.3 Pattern matching

Grazie al pattern matching è possibile effettuare complesse operazioni di estrazione dei dati e manipolazione degli stessi, è un aspetto centrale di ℂDuce e ne incrementa moltissimo la potenza espressiva. Purtroppo non è lo strumento più semplice da padroneggiare e a una prima vista può sembrare ambiguo il modo in cui descrivere un certo pattern. Lievi differenze nella definizione del pattern producono effetti differenti, e bisogna prestare particolare attenzione a descrizioni che apparentemente possono sembrare equivalenti.

## 5.3 Spunti per paragoni futuri

In questo lavoro abbiamo provato a fare dei paragoni tra ℂDuce e Protégé, sono due strumenti molto differenti e le scale di paragone, a seconda dei parametri considerati, fanno risaltare uno strumento come molto efficace e l'altro come inappropriato. Potrebbe essere quindi interessante confrontare ℂDuce con altri strumenti più simili.

### 5.3.1 Confronto con un linguaggio imperativo

Esistono vari linguaggi che incorporano librerie per la modellazione di documenti XML, uno fra tutti è sicuramente java.

#### JAXP

JAXP [12] è l'API che java mette a disposizione per processare dati in XML. I packages contenuti in questa API consentono alle applicazioni di fare il parsing, trasformare, validare e interrogare con delle Query documenti XML. JASP è dotata di un layer che permetta l'indipendenza tra il codice dell'applicazione e il particolare processor XML implementato. JAXP, rispetto a ℂDuce che è attualmente in stato di sviluppo ed è considerato un prototipo, garantisce la possibilità di sviluppare applicazioni per il processing di file XML che siano perfettamente funzionanti, integrabili e stabili.

Alla luce dei vantaggi di affidabilità offerti da progetti molto più grandi di ℂDuce potrebbe essere interessante valutare come l'uso di un linguaggio funzionale possa apportare benefici a livello di programmazione, correttezza del codice e leggibilità rispetto a un linguaggio essenzialmente imperativo come java.

### 5.3.2 Confronto con un linguaggio funzionale

ℂDuce nasce con lo scopo di processare documenti XML, esistono altri linguaggi più general porpouse che implementano la capacità di elaborare XML tramite librerie. Potrebbe essere interessante confrontare ℂDuce con un altro linguaggio funzionale per valutare se e in che modo la specificità di ℂDuce risulta essere un vantaggio oppure un limite allo sviluppo di applicazioni.

Uno dei linguaggi che presenta ampia scelta di librerie è Haskell. Questo linguaggio è sicuramente più noto di ℂDuce e sono disponibili, pertanto, numerose risorse online per chiarire domande e dubbi che possono sorgere. Fra le librerie per l'elaborazione di documenti XML citiamo HaXml[23] e Haskell XML Toolbox (HXT)[24], che si basa sulle idee di HaXml, entrambi i progetti risultano attivi e documentati[25].

Le librerie che abbiamo citato non sono state valutate approfonditamente, ne diamo quindi una descrizione molto breve: permettono di trattare in modo più naturale le strutture ad albero dei documenti XML rispetto a quello che sarebbe necessario fare in Haskell puro.

Il vantaggio di usare una di queste librerie è la possibilità di interfacciarsi direttamente con Haskell potendo integrare le funzionalità della libreria scelta con le numerosissime funzioni offerte da Haskell (sia prese dai pacchetti base sia implementate da librerie esterne). Lo svantaggio sta nella minor naturalezza con cui si descrive la struttura del file XML, queste librerie infatti non raggiungono la stessa naturalezza e concisione espressiva delle descrizioni degli elementi in ℂDuce.

Il confronto si potrebbe allora concentrare su quanto pesino i vantaggi e gli svantaggi derivanti dall'uso di una libreria specifica in un linguaggio general-porpouse rispetto all'uso di uno strumento sviluppato ad-hoc.

Per quanto riguarda l'integrazione di ℂDuce con un altro linguaggio funzionale in modo da usare librerie e funzioni esterne abbiamo descritto nel paragrafo 5.1.3 le difficoltà riscontrate cercando di unire ℂDuce con OCaml.

### 5.3.3 Confronti possibili

Abbiamo presentato altri strumenti alternativi a ℂDuce, i confronti possibili potrebbero basarsi sulle seguenti caratteristiche:

- velocità di sviluppo del codice: ci aspettiamo che un linguaggio funzionale possa presentare dei vantaggi soprattutto nell'esprimere delle trasformazioni degli elementi

[23] https://archives.haskell.org/projects.haskell.org/HaXml/

[24] https://wiki.haskell.org/HXT

[25] Per una lista esaustiva delle librerie e funzioni per trasformare documenti XML in Haskell si può cercare la parola chiave XML su https://hoogle.haskell.org/ e https://hackage.haskell.org/

rispetto a un linguaggio imperativo; sfruttando poi la funzione `map` (sia in ℂDuce che Haskell) risulta anche immediato applicare la trasformazione a intere liste di elementi;

- correttezza del software: anche in questo caso un linguaggio funzionale che esegua un controllo statico dei tipi in fase di compilazione può aiutarci a evidenziare errori prima che si presentino a run-time;
- scalabilità e integrazione: probabilmente da questo punto di vista un linguaggio come java offre dei vantaggi essendo pensato anche per lo sviluppo di applicativi di grandi dimensioni più che Haskell o ℂDuce (in ogni caso nulla vieta di integrare ℂDuce in applicazioni scritte in altri linguaggi, questa possibilità è approfondita nel paragrafo 5.4.2);
- velocità di esecuzione: durante gli esperimenti si è notato un drastico rallentamento dell'esecuzione quando si usavano funzioni ricorsive per creare gli insiemi di sottoclassi, può essere utile provare almeno in java a implementare una versione iterativa dello stesso algoritmo per verificare se le prestazioni cambiano significativamente. Per quanto riguarda le Query può essere interessante un confronto dato che queste ultime in ℂDuce sono ottimizzate con le stesse tecniche di ottimizzazione della logica classica SQL.

## 5.4 Sviluppo di nuovi strumenti

### 5.4.1 Criticità di ℂDuce

ℂDuce si è rivelato essere un valido e potente strumento per la manipolazione di file XML; a prescindere dai problemi riscontrati durante gli esperimenti, che auspicabilmente verranno tutti risolti con il rilascio della nova versione stabile. La più grande difficoltà che un utente incontra quando si approccia all'uso di questo strumento è proprio la difficoltà di esprimere degli algoritmi con un linguaggio funzionale.

Chi vuole usare ℂDuce per manipolare delle ontologie non ha grandi competenze in campo di programmazione e ha usato fino a quel momento solo strumenti grafici per la creazione o manipolazione di ontologie si scontra con una notevole difficoltà di sviluppo degli strumenti, che solo in parte sono guidate dal tutorial presente sul sito, in numerosi casi bisogna essere in grado di interpretare la guida che a un utente non esperto può sembrare poco chiara.

Discorso analogo vale per un utente già competente nell'uso di linguaggi imperativi che si trova a dover fare i conti con un nuovo paradigma di programmazione senza che la guida o il tutorial aiutino particolarmente. Probabilmente un utente di questo tipo si troverà ancora più confuso di un utente novizio alla programmazione.

ℂDuce presuppone che i suoi utenti posseggano delle competenze già abbastanza avanzate nell'uso di linguaggi funzionali, che si trovino a proprio agio nello sviluppare algoritmi ricorsivi e a usare intensivamente il pattern matching. Date le competenze richieste e la curva d'apprendimento particolarmente ripida ℂDuce potrebbe rivelarsi uno strumento ostico per la maggior parte degli utenti che vogliono concentrarsi sulla creazione di un'ontologia che sia corretta, espressiva e ben formata e non sullo sviluppo di codice per la creazione stessa.

### 5.4.2 Interfaccia grafica

Fatte le precedenti considerazioni ne risulta che nonostante ℂDuce sia uno strumento potente ed espressivo sia anche particolarmente poco fruibile. Per arginare la difficoltà d'uso si potrebbe pensare a un'interfaccia che permetta di rendere grafiche la maggior parte delle

operazioni, relegando la parte di programmazione a task specifici o che operano ad hoc sulla particolare ontologia che si sta manipolando.

Esistono numerose interfacce da cui trarre ispirazione[26] (l'interfaccia di Protégé per citarne una) e ℂDuce si presta particolarmente bene a creare del codice generale:

- parsing di una generica ontologia: come si può notare nel listato 4.1 la struttura definita permette di fare correttamente il parsing di tre ontologie completamente differenti, la struttura non è esaustiva e si potrebbero dover aggiungere altre definizioni di tipi per renderla completamente generale senza rinunciare ai controlli di correttezza[27]; questa aggiunta risulta, in ogni caso, fattibile e non dispendiosa;

- funzioni generali per trattare con ontologie: guardando il codice del listato 4.2 ci si rende conto che queste funzioni non hanno alcuna attinenza con le particolari ontologie che stiamo trattando, un parco sufficientemente ampio di queste funzioni sarebbe in grado di estrarre la maggior parte delle informazioni utili da un'ontologia, di selezionare e di assemblare le parti che l'utente vorrebbe poter mantenere;

- funzioni di ordine superiore: laddove le funzioni generali non bastano, si potrebbero pensare funzioni di ordine superiore complesse a piacere che servano per manipolare in modo molto specifico le ontologie che si stanno trattando: queste funzioni vengono messe a disposizione dell'utente che deve solo integrare piccole e semplici funzioni che, prese come argomento dalla funzione di ordine superiore, la rendano specifica per la particolare ontologia d'interesse. Questo limita notevolmente la necessità di programmare dell'utente anche quando si voglia costruire degli strumenti molto specifici per la propria ontologia;

- creazione guidata di Query: esistono già numerosi strumenti che permettono la creazione di Query, anche particolarmente complesse, in modo grafico e guidato; data la profonda somiglianza tra le Query espresse in linguaggio SQL e la sintassi in ℂDuce, sarebbe possibile riproporre uno strumento del genere per rendere le Query accessibili.

### 5.4.3 Confronto con esperti

Un passo importante prima di procedere nello sviluppo di nuovi strumenti e di un'interfaccia grafica, sarà proporre i risultati ottenuti a esperti del settore della rappresentazione della conoscenza.

Da tale confronto potranno nascere interessanti considerazioni sull'uso di ℂDuce nella manipolazione delle KB, in particolare nel confronto bisognerà valutare:

- se effettivamente i risultati raggiunti possono essere interessanti e utili dal punto di vista dello sviluppo e della manipolazione delle basi di conoscenza;

- se le garanzie di correttezza offerte de ℂDuce sono effettivamente importanti nella realizzazione di KB consistenti e non contraddittorie e se questo vantaggio in termini di correttezza controllata staticamente è sufficiente a giustificare il fatto che gli utilizzatori possano dover implementare delle funzioni per sfruttare appieno le potenzialità di ℂDuce;

- se, infine, i due punti precedenti mettono in luce che ℂDuce è un buono strumento per la manipolazione della conoscenza formale allora l'esperienza e la competenza

[26] Se la creazione di una nuova interfaccia da zero rappresenta un ostacolo eccessivo, si può pensare di costruire in ℂDuce dei plug-in per Protégé.

[27] Se si usasse il tipo `AnyXml` la struttura sarebbe generalissima ma non offrirebbe più controlli puntuali sulla correttezza dei tipi restituiti dalle funzioni.

di un esperto della rappresentazione della conoscenza sarebbero fondamentali per poter ideare e sviluppare un'interfaccia grafica che possa essere funzionale, comoda ed efficiente da utilizzare.

## 5.5 Conclusioni

ℂDuce nonostante sia classificato come prototipo, è uno strumento completo ed espressivo per la manipolazione di documenti XML e in particolare di ontologie[28]. Offre una documentazione online ben curata e accessibile a un utente con solide basi di programmazione funzionale; la documentazione è inoltre accompagnata da una serie di esempi che guidano, almeno fino a un certo punto, un utente più novizio a muovere i primi passi con lo strumento potendo già realizzare programmi utili.

Paragonandolo ad altri strumenti, si rivela acerbo e il sito riflette questa condizione essendo finito solo per metà (numerosi link e componenti non sono attualmente funzionanti). Nonostante questo, rispetto a software molto più grandi e più specificatamente orientati alla manipolazione di ontologie, ℂDuce permette di esprimere una grande quantità di lavoro in poche righe di codice, in modo elegante e leggibile.

ℂDuce si è dimostrato uno strumento efficace nella manipolazione di ontologie e, anche se non è uno strumento specifico per questo scopo, si dimostra, in certi casi d'uso, più potente di Protégé. In particolare l'elaborazione di molti dati, la selezione secondo certi criteri e la trasformazione di elementi seguendo certi pattern risulta molto naturale in questo linguaggio. I vantaggi appena elencati e il controllo statico su tipi che garantisce la costruzione di ontologie ben formate rendono ℂDuce un valido strumento per la manipolazione delle KB.

In questo elaborato ci siamo concentrati su 2 casi d'uso, che, per quanto rilevanti, non coprono l'intero mondo della rappresentazione formale della conoscenza. Dati i risultati ottenuti con ℂDuce in questi ambiti, però, lo si ritiene meritevole di ulteriore analisi, anche con esperti del settore, perché potrebbe rivelarsi uno strumento in grado, se non di sostituire, quantomeno di affiancare software per l'editing di KB molto più noti apportando numerosi vantaggi nella manipolazione dei dati.

[28] Si fa riferimento alla versione del software 0.6.1-rc1 .

# Elenco listati

# Bibliografia


[1] Grigoris Antoniou e Frank van Harmelen. "Web ontology language: Owl". In: *Handbook on ontologies* (2009), pp. 91–110.

[2] Sean Bechhofer e Alistair Miles. "Using OWL and SKOS". In: *W3C. Web page* (2008).

[3] Véronique Benzaken, Giuseppe Castagna e Alain Frisch. "CDuce: an XML-centric general-purpose language". In: *ACM SIGPLAN Notices* 38.9 (2003), pp. 51–63.

[4] Tim Berners-Lee, James Hendler e Ora Lassila. "A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities". In: *Scientific american* 284.5 (2001), pp. 34–43.

[5] Tim Bray, Dave Hollander, Andrew Layman e Richard Tobin. "Namespaces in XML". In: *World Wide Web Consortium Recommendation REC-xml-names-19990114. http://www. w3. org/TR/1999/REC-xml-names-19990114* (1999).

[6] Dan Brickley, Ramanathan V Guha e Andrew Layman. *Resource description framework (RDF) schema specification*. Rapp. tecn. Technical report, W3C, 1999. W3C Proposed Recommendation. http://www. w3 . . ., 1998.

[7] World Wide Web Consortium et al. "Extensible markup language (XML) 1.1". In: (2006).

[8] Thomas R Gruber. "A translation approach to portable ontology specifications". In: *Knowledge acquisition* 5.2 (1993), pp. 199–220.

[9] Pascal Hitzler. "A review of the semantic web field". In: *Communications of the ACM* 64.2 (2021), pp. 76–83.

[10] Haruo Hosoya e Benjamin C Pierce. "XDuce: A statically typed XML processing language". In: *ACM Transactions on Internet Technology (TOIT)* 3.2 (2003), pp. 117–148.

[11] Antoine Isaac e Ed Summers. "SKOS simple knowledge organization system primer". In: *Working Group Note, W3C* (2009).

[12] *Java API for XML Processing*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxp/index.html.

[13] Adam Kilgarriff e Colin Yallop. "What's in a Thesaurus?" In: *LREC*. 2000, pp. 1371–1379.

[14] Daniel Kless, Ludger Jansen, Jutta Lindenthal e Jens Wiebensohn. "A method for re-engineering a thesaurus into an ontology." In: *FOIS*. 2012, pp. 133–146.

[15] Jian Bing Li e James Miller. "Testing the semantics of W3C XML schema". In: *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*. Vol. 1. IEEE. 2005, pp. 443–448.

[16] Frank Manola, Eric Miller, Brian McBride et al. "RDF primer". In: *W3C recommendation* 10.1-107 (2004), p. 6.

[17] Alistair Miles e Sean Bechhofer. “SKOS simple knowledge organization system reference”. In: *W3C recommendation* (2009).

[18] E MILLER. “An introduction to the resource description framework”. In: *Bulletin of the American Society for Information Science* 25.1 (1998), pp. 15–19.

[19] *Protégé 5 Documentation*. URL: https://protegeproject.github.io/protege/getting-started/.

[20] Stuart J Russell e Peter Norvig. *Artificial Intelligence A Modern Approach*. London, 2010. Cap. 7, p. 235.

[21] R Sivakumar e PV Arivoli. “Ontology visualization PROTÉGÉ tools a review”. In: *International Journal of Advanced Information Technology (IJAIT) Vol* 1 (2011).

[22] Padmini Srinivasan. “Thesaurus construction”. In: *Information Retrieval: data structures and algorithms* (1992), pp. 161–218.

[23] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez e Mariano Fernandez-Lopez. “The NeOn Methodology framework: A scenario-based methodology for ontology development”. In: *Applied ontology* 10.2 (2015), pp. 107–145.

[24] World Wide Web Consortium (W3C). *XML - Extensible Markup Language*. URL: https://www.w3.org/XML/.