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 Novembre 21, 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 è 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 di fare le stesse modifiche con strumenti grafici.

Indice

Capitolo 1

Concetti di base

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.

1.1 Ontologie e tesauri

1.1.1 ontologie

Esempio

Consideriamo una semplice ontologia che rappresenta persone con legami di parentela genitore-figlio; le persone hanno uno o più nomi salvate 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>
        </owl:equivalentClass>
        <rdfs:subClassOf rdf:resource="http://www.persone#Persone"/>
    </owl:Class>
    <!-- http://www.persone#Persone -->
    <owl:Class rdf:about="http://www.persone#Persone"/>
    <!-- http://www.persone#Bruto -->
    <owl:NamedIndividual rdf:about="http://www.persone#Bruto">
        <rdf:type rdf:resource="http://www.persone#Persone"/>
        <rdfs:comment>Bruto</rdfs:comment>
    </owl:NamedIndividual>
    <!-- http://www.persone#Cesare -->
    <owl:NamedIndividual rdf:about="http://www.persone#Cesare">
        <rdf:type rdf:resource="http://www.persone#Genitori"/>
        <www:genitoreDi rdf:resource="http://www.persone#Bruto"/>
        <rdfs:comment>Caio</rdfs:comment>
        <rdfs:comment>Augusto</rdfs:comment>
        <rdfs:comment>Giulio</rdfs:comment>
        <rdfs:comment>Cesare</rdfs:comment>
    </owl:NamedIndividual>
</rdf:RDF>

```

1.1.2 Tesauri

1.2 Strumenti di editing

1.2.1 Protégé

1.2.2 ℂDuce

1.2.3 Feature

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.

1.3 Metalinguaggi

1.3.1 OWL

1.3.2 SKOS

Capitolo 2

Strumenti offerti da ℂDuce

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 vedendo dei primi comandi per trattare ontologie dalla struttura semplice.

2.1 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. Ad ogni elemento del vettore sarà associato un tipo che avrà la stessa struttura del tipo generale e permetterà di descrivere la struttura del documento XML discendendo fino alle foglie.

2.1.1 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 correttamente interpetare 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; Al 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 2 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 documento 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.

2.2 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 come descritto in 1.2.3
- `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.

2.2.1 Esempio

Consideriamo nuovamente l'ontologia padri-figli con struttura definita in 2.1 e costruiamo 2 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, lego la variabile `n` alla sequenza di tutti gli elementi di tipo `Name` associati a quell'individuo, poi uso `map` per estrarre da ogni elemento di questa lista solo la stringa col nome.

Nella seconda funzione uso `transform` per selezionare dalla lista `x` (lista di elementi tipo `Thing`) solo gli elementi `Individual`, a questi applico la funzione `name` sopra definita. La seconda clausola di `transform`, che serve a buttare via 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` ad 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.

2.3 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 `b` 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 b 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 ad 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

2.3.1 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`.

Gli ultimi due esempi 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 non è così il risultato sarà sempre vuoto.

Il secondo errore è dovuto al fatto che mettendo le parentesi quadre intorno a `sel` questo viene interpetato come una sequenza (anche se lo è già), ne risulta una sequenza il cui unico elemento è una sequenza (in paticolare 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 che in effetti restituisce il risultato atteso.

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. Nei capitoli 3 e 4 applicheremo gli strumenti analizzati ad 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

Introduzione

Qui vediamo come sia possibile usare ℂDuce per creare ontologia a partire da una rappresentazione della conoscenza già formalizzata in altri modi. Per illustrare il processo di creazione faremo riferimento ad 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 [2]). 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 vediamo brevemente quali altri strumenti avremmo potuto utilizzare, in particolare cercheremo di ottenere risultati analoghi con Protégé discutendo vantaggi e svantaggi di ciascun approccio.

3.1 Vantaggi del passaggio da tesauro ad ontologia

Potremmo essere interessati a trasformare un tesauro espresso in SKOS in un'ontologia espessa 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.

3.1.1 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 è pubblico può essere scaricato liberamente da chiunque sia interessato. Il tesauro permette di accedere in modo logico, organizzato e strutturato

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

ad una vasta conoscenza. Data la struttura ad albero le relazioni tra oggetti sono chiare e l'aggiunta o la ricerca di informazioni può essere fatta in modo rapido ed efficiente.

3.1.2 Svantaggi del tesauro

Data la sua struttura ad albero 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 che una bandana è un accessorio per il capo, perché quest'ultima non si trova nel ramo degli accessori della testa.

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

3.2 Struttura del tesauro

La struttura del tesauro che vogliamo re-ingegnerizzare è semplice, in particolare abbiamo quattro concetti fondamentali da cui partono tutti gli altri rami che rappresentano i concetti derivati. I concetti fondamentali sono:

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

Ogni concetto che sia uno di quelli fondamentali o meno 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 "The Getty Vocabularies"[3].

Analizzata la stuttura 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> [];;
```

3.3 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 ad 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 molto semplificata rispetto alla struttura del listato 2.1, questo perché adesso ci interessa solo ricostruire con classi OWL i concetti SKOS.

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 [1]).

Si potrebbero anche eliminare completamente 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.

3.4 Da concetto SKOS a classe OWL

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

intero concetto del tesauro ad una classe dell'ontologia; infine, usando la funzione `map` (2.2), possiamo applicare questa funzione a tutti i concetti del tesauro per ottenere le classi che popoleranno l'ontologia.

3.4.1 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 e 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 2 valori (`Concept` e `ConceptScheme`). Le funzioni possono essere scritte in ℂDuce come segue:

Listato 3.3: SKOS_to_OWL.cd

```
let fun transformNote (ScopeNote -> Note)
  x -> x

let fun transformDictionary (ExactMatch -> Dictionary)
  x -> x

let fun transformLabel (PrefLabel -> Label ; AltLabel -> Label)
  | <skos:prefLabel xml:lang=l> lab -> <rdfs:label xml:lang=l> lab
  | <skos:altLabel xml:lang=l> lab  -> <rdfs:label xml:lang=l> lab;;

let fun transformSubClass (Broader -> SubClass)
  <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, vediamo 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`; una volta legate queste variabili le usiamo 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, leghiamo un'unica variabile `res` alla stringa che identifica il nodo padre e con questa costruiamo il tag `subClassOf`.

3.4.2 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 * ]`) per fare questo usiamo le funzioni definite prima (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

```
let fun transformAtt (DescAtt -> [ClassAtt*])
  | x & PrefLabel  -> [(transformLabel x)]
  | x & AltLabel   -> [(transformLabel x)]
  | x & ScopeNote  -> [(transformNote x)]
  | x & ExactMatch -> [(transformDictionary x)]
  | x & Broader    -> [(transformSubClass x)]
  | Any            -> [];;

let fun transformClass (Desc -> Class)
  <rdf:Description rdf:about=ab> [ (descAtt :: DescAtt)* ] ->
    let classAtt = flatten ( map descAtt with x -> transformAtt x) in
    <owl:Class rdf:about=ab> classAtt;;
```

Le funzioni sono abbastanza semplici, descriviamo brevemente la seconda: 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 la 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.

3.5 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

```
let fashion :? Thesaurus = load_xml "thesaurus.rdf";;

let newClass = map [fashion]/Desc with x -> transformClass x
in
let newOnt : Ontology = <rdf:RDF
↪  xml:base="http://www.semanticweb.org/OntEur"> ( [ <owl:Ontology
↪  rdf:about="OntEur"> [ ] ] @ newClass )
in
dump_to_file_utf8 "Ontologia.rdf" (print_xml_utf8 newOnt);;
```

Alla linea 1, facendo riferimento alla struttura del tesauro definita in 3.1, carichiamo il tesauro; alla riga 2 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 ad ognuno di questi elementi la funzione `transformClass` (3.4), infine diamo un nome al vettore di classi appena creato in modo da poterlo usare nel segiuto 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.

3.6 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 rispetti 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` per trasformare gli attributi da SKOS a OWL per ogni elemento estratto con la funzione `map`.

3.7 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 (**??**) presenteremo un'estensione della struttura dell'ontologia definita qui 3.2 in cui andremo ad arricchire con nuove informazioni l'ontologia qui creata ed useremo la nova versione per sviluppare selezioni e trasformazioni più sofisticate di quelle che avremo potuto definire su un tesauro.

3.8 In Protégé

Provando ad 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 4 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. Senza voler operare in modo puntuale su ogni individuo per trasformarlo manualmente in una classe (e senza usare i plug-in che permettono di scrivere programmi, ad esempio in java[4], per operare delle trasformazioni simili a quelle fatte con ℂDuce) ciò che si può fare è comunque definire le stesse relazioni che abbiamo accennato nel paragrafo precedente (3.7) 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 che le relazioni che valorizzeremo siano corrette (ad esempio dato dominio e range della relazione "è fatto di" 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, al posto che con la più naturale definizione di sottoclasse.

3.9 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 [2] 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 [3] 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 ad ontologia non risulta particolarmente complessa o lunga in termini di tempo (come testimonia la funzione 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 venuti a galla fino al momento dell'esecuzione, in particolare dovendo lavorare alternativamente su vettori ed elementi singoli è capitato spesso che il tipo inferito fosse un vettore, mentre il tipo richiesto fosse un elemento o viceversa (in un linguaggio come C con l'uso dei puntatori un errore del genere avrebbe potenzialmente richiesto molto tempo per essere individuato e risolto).

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

Capitolo 4

Merge di ontologie

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 ad 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.

4.1 Obiettivo del merge

Adesso che abbiamo creato un ontologia a partire dall'"European Fashion Thesaurus" e abbiamo descritto una semplice ontologia per descrivere persone vorremmo fonderle in una nuova ontologia che possa parlare degli usi e costumi della società, magari riferita ad 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 uno status sociale, un lavoro e dell'abbigliamento alle persone.

4.1.1 Ontologie di partenza

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

4.1.2 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 artificiali 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`.

4.2 Struttura generica di un'ontologia

Per fare il merge dobbiamo poter caricare le 3 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 2 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 thesauro i tag SKOS sono stati correttamente riconosciuti [1] e classificati come tag `owl:AnnotationProperty`;

4.3 Merge

4.3.1 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 la particolare ontologia 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))
```

```
28    | [] -> `false;;
29
30  let fun classOf (Individual -> Ontology -> [Class*])
31    <owl:NamedIndividual ..> [ (tp::IndClass | _) *] -> fun (Ontology ->
      ↪  [Class*])
32    ont ->
33      transform tp with <rdf:type rdf:resource=str> [] ->
34        transform [ont]/Class with x ->
35          match x with <owl:Class rdf:about=a>[ _* ] -> if a = str then [x]
            ↪  else [];;
36
37  let fun contains (Any -> [Any*] -> Bool)
38    obj -> fun ([Any*] -> Bool)
39    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 a applicazioni parziali (curried functions[1]), questo è utile per poter parametrizzare anche l'ontologia di riferimento per l'operazione; 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` 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 la coda è vuota (seconda clausola del matching) si 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).

[1] https://en.wikipedia.org/wiki/Currying

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 delle 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 ad essere impegnativo dal punto di vista implementativo è anche poco efficiente dal punto di vista prestazionale. Come vedremo in 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` che prende un individuo e una lista di classi, valuta a che classi appartiene l'individuo 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`

4.3.2 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 risulterebbero una ridondanti a livello di presentazione se confrontate con le funzioni che descriviamo ora.

Per ragionare sul singolo abito ci domandiamo di che materiali è fatto usando la relazione `made_of` che associa uno o più materiali ad 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[2].

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

Listato 4.3: test artificial

[2]possiamo fare inferenza solo coi 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

```
let fun madeOf (Individual -> Ontology -> [Individual*])
  <owl:NamedIndividual ..> [ (ip::IndProp | _) *] -> fun (Ontology ->
  ↪ [Individual*])
  ont ->
    transform ip with <ns1:made_of rdf:resource=str> [] ->
      transform [ont]/Individual with x ->
        match x with <owl:NamedIndividual rdf:about=a>[ _* ] -> if a = str
        ↪ then [x] else [];;


let fun isArtificial (Individual -> [Class*] -> [Class*] -> Ontology ->
↪ Bool)
  ind -> fun ([Class*] -> [Class*] -> Ontology -> Bool)
  materials -> fun ([Class*] -> Ontology -> Bool)
  artificialMats -> fun (Ontology -> Bool)
  ont -> if (isInClasses ind materials ont)
    then (isInClasses ind artificialMats ont)
    else
      let matMade = madeOf ind ont in
      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 2 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 ad 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.

4.3.3 Costruzione dell'ontologia

Assembliamo tutti i pezzi visti fin'ora 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

```
let fashion = loadOntology "europeana_formatted.rdf";;
let people = loadOntology "people.rdf";;
let society = loadOntology "society.rdf";;

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 riga 1 a riga 3 carichiamo le ontologie che useremo, poi andiamo a creare le liste di classi di materiali che ci servono per utilizzare le funzioni definite in 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 riga 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 riga 24 e la classe `people` andrà modificata per renderla equivalente alla classe `people` dell'ontologia `people` (riga 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 riga 26 e alla riga 28 costruiamo la nuova ontologia, in riga 30 la salviamo su file.

4.3.4 Risultato finale

L'ontologia che abbiamo costruito permette di descrivere la società nelle modalità che ci eravamo prefissati all'inizio del capitolo (4.1), 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.

4.4 In Protégé

In potégé 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).

Nel caso in cui volessimo fare una selezione più fine su che concetti importare possiamo, una volta fatto il merge, dovremmo procedere in modo differente, o rimuovendo ciò che non ci interessa o modificando le ontologie di partenza costruendo a prescindere (mediante delle query) le classi che vogliamo importare.

4.5 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 ad essere consistenti e che le informazioni non vengano alterate durante il merge. ℂDuce viene incontro al programmatore con un potente controllo sul tipo restituito dalle funzioni, in questo caso possiamo notare che man mano che creiamo la nuova ontologia in 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` (riga 20 in 4.4) non era andata a buon fine e aveva generato un elemento XML che non faceva più il match con l'elemento di tipo `Class` definito in 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 in 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 in 4.4 che assemble 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 alti, ℂDuce si rivela più efficacie 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 vanga 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 loro pagina GitLab[1] il progetto 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 in 1.2.3. L'implementazione attuale ha lo scopo di mostrare proprio queste feature innovative e di validare le scelte di progettazione effettuate.

5.1.2 Stato di sviluppo attuale

Sulla pagina del progetto[6] le ultime informazioni risalgono al 2021 e spiegano che il 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 ulime 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[7].

[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] https://www.cduce.org/
[7] alla data in cui si scrive (10/10/2023) l'ultima attività risulta del 09/10/203

5.1.3 Difficoltà incontrate

Ho testato ℂDuce su una distribuzione Ubuntu[8] e su Arch[9]; 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 ad oggi i pacchetti di OCaml sono stati ulteriormente aggiornati, e il workaround descritto non è più sufficiente. Ho provato, questa volta con successo, ad 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 ad usare ℂDuce , sono presenti in particolare 2 documenti:

- una guida utente[10] 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 estensivo uso 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[11] che parte direttamente da degli 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 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

[8] https://ubuntu.com/desktop
[9] i use Arch btw: https://archlinux.org/
[10] https://www.cduce.org/manual.html
[11] https://www.cduce.org/tutorial.html

tutorial. Avendo delle basi di Haskell[12] 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` in 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 2 passi: prima trasformiamo l'elemento di tipo `AnyXml` (che contiene il file XML che vogliamo salvate) in una stringa poi facciamo il dump della stringa su file. Per la conversione da XML a stringa abbiamo 2 possibilità, se il documento contiene solo caratteri previsti dalla norma ISO-8859-1[13] 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 tende 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 interpetato correttamente, ma se vogliamo vedere il risultato della manipolazione con ℂDuce senza ricorrere ad altri software appositi ci troviamo in difficoltà[14]. 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 operazioni di output 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.

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 però un programma in java per essere fruito.

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 ed 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 ad interessarsi al 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[15].

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

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

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

[14] 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 imossibile conoscere il documento originale

[15] https://gitlab.math.univ-paris-diderot.fr/cduce/cduce/-/wikis/home

funzionale lo rende particolarmente sintetico e leggibile una volta presa dimestichezza coi concetti base.

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 ad 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 debug può 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 un una funzione per la rimozione selettiva che fra i vari parametri accetta 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 in 5.4

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 ad 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 per forza, 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.

Elenco listati

Bibliografia

[1] Sean Bechhofer e Alistair Miles. *Using OWL and SKOS*. 2008. URL: https://www.w3.org/2006/07/SWD/SKOS/skos-and-owl/master.html.

[2] Daniel KLESS, Ludger JANSEN, Jutta LINDENTHAL e Jens WIEBENSOHN. *A method for re-engineering a thesaurus into an ontology*. 2012. URL: https://jansenludger.github.io/home/Texte/Kless%20et%20al_A%20method%20for%20reengineering%20a%20thesaurus%20into%20an%20ontology%20_FOIS%202012_Preprint.pdf.

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