Sistemi Basati su Conoscenza e Ontologie

Nicola Fanizzi
Ingegneria della Conoscenza
CdL in Informatica • *Dipartimento di Informatica*
Università degli studi di Bari Aldo Moro

indice

Implementare Sistemi Basati su Conoscenza

Riflessione (*reflection*)

Capacità di rappresentare il proprio ragionamento e ragionare su di esso:

- consente la customizzazione dei KBS per esigenze particolari

Si considererà l'*implementazione* di strumenti leggeri per costruire nuovi *linguaggi* con caratteristiche richieste da particolari *applicazioni*:

- facilitandone l'implementazione si facilita la loro *adozione*
- linguaggio e strumenti possono *evolvere* con l'applicazione

Meta-Interpreti

Meta-interprete: interprete scritto nello stesso linguaggio interpretato

- vantaggio: *prototipazione rapida* di linguaggi con nuove caratteristiche
  - per aumentarne l'efficienza si può successivamente realizzare un *compilatore*
- *Nomenclatura*
  - linguaggio implementato: **linguaggio-base** (o ling.-*ground*, o ling.-*oggetto*)
    - per espressioni a **livello-base**
  - linguaggio nel quale lo si implementa: **meta-linguaggio**
    - per espressioni di **meta-livello**

Agenda - Si considereranno:

- un meta-interprete per il linguaggio delle clausole definite
- modifiche / estensioni del linguaggio-base
- modificando il meta-interprete, strumenti per la spiegazione e il debugging

Serve rappresentare espressioni di livello-base
manipolabili dal meta-interprete per produrre risposte:
linguaggio delle *clausole definite*

- Il meta-linguaggio deve potersi riferire agli elementi sintattici del ling.-base:
  - simboli di meta-livello: denotano *termini* del livello-base
    - atomi e clausole
  - termini livello-base: denotano *oggetti* nel dominio da modellare
  - predicati del livello-base: denotano relazioni in tale dominio

Rappresentazioni Alternative

Per i termini:

- **rappresentazione non-ground**: stesso tipo di termine per i due livelli
  - → variabili a livello-base rappresentate come variabili nel meta-livello
  - *unificazione* nel meta-livello per unificare termini di livello-base
  - es. termine $foo(X, f(b), X)$ livello-base → $foo(X, f(b), X)$ meta-livello
- **rappresentazione ground**: variabili del linguaggio-base rappresentate come costanti nel meta-linguaggio
  - consente modelli di *unificazione* più sofisticati
  - es. termine $foo(X, f(b), X)$ livello-base → $foo(x, f(b), x)$ meta-livello
    - ma x va anche dichiarata come variabile di livello-base

nel seguito: rappresentazione *non-ground*

Rappresentazione non-ground: basi

Nella rappresentazione *non-ground*:
meta-linguaggio per rappresentare tutti i costrutti di livello-base

- *variabili*, *costanti* e *funzioni* di livello-base come variabili, costanti e funzioni corrispondenti di meta-livello
  - *stesso* tipo di *termine* per entrambi i livelli
- simbolo di *predicato* p a livello-base come simbolo di *funzione* p nel meta-livello
- *atomo* $p(t_1, \ldots, t_k)$ a livello-base come *termine* $p(t_1, \ldots, t_k)$ nel meta-livello
- *corpi* (congiunzioni di atomi) a livello-base come *termini* nel meta-livello
  - dati e_1 e e_2 *termini* di meta-livello per *atomi* (o loro congiunzioni) a livello-base
    → $oand(e_1, e_2)$ *termine* di meta-livello per la loro *congiunzione*
    - $oand$ funzione (meta) per l'op. di congiunzione (base)

Rappresentazione Non-ground: Clausole

clausole definite a livello-base come atomi/fatti nel meta-livello

- *regola* $h \leftarrow b$ rappresentata con $clause(h, b')$
  - b' rappresentazione del corpo b
- *fatto* rappresentato con $clause(a, true)$
  - costante di meta-livello $true$ denota il corpo vuoto a livello-base

---

Esempio — clausole viste in precedenza:

- livello-base:
  - $connected\_to(l_1, w_0).$
  - $connected\_to(w_0, w_1) \leftarrow up(s_2).$
  - $lit(L) \leftarrow light(L) \wedge ok(L) \wedge live(L).$
- fatti corrispondenti nel meta-livello:
  - $clause(connected\_to(l_1, w_0), true).$
  - $clause(connected\_to(w_0, w_1), up(s_2)).$
  - $clause(lit(L), oand(light(L), oand(ok(L), live(L)))).$

Rappresentazione Non-ground: Notazione Alternativa

Notazione *infissa* per aumentare la *leggibilità*:

- $e_1 \,\&\, e_2$ in vece di $oand(e_1, e_2)$
  - il simbolo di *funzione* $\&$ nel meta-linguaggio denota l'operatore infisso $\wedge$, tra atomi (o loro congiunzioni) nel linguaggio-base
- $h \Leftarrow b$ in vece di $clause(h, b)$
  - op. $\leftarrow$ rappresentato al meta-livello con il simbolo di *predicato* $\Leftarrow$ (infisso)
  - *clausola* a livello-base $h \leftarrow a_1 \wedge \cdots \wedge a_n$ come *atomo* di meta-livello $h \Leftarrow a_1 \,\&\, \cdots \,\&\, a_n$
    - vero se la corrispondente clausola è parte della KB a livello-base

---

Esempio — *clausole* precedenti come *fatti*:

- $connected\_to(l_1, w_0) \Leftarrow true.$
- $connected\_to(w_0, w_1) \Leftarrow up(s_2).$
- $lit(L) \Leftarrow light(L) \,\&\, ok(L) \,\&\, live(L).$

Rappresentazione Non-ground: Riepilogo

Rappresentazione *non-ground* per il linguaggio-base nel meta-livello:

livello-base	meta-livello
variabile X	variabile X
costante c	costante c
simbolo di funzione f	simbolo di funzione f
simbolo di predicato p	simbolo di funzione p
op. `and` $\wedge$	simbolo di funzione $\&$
op. `if` $\leftarrow$	simbolo di predicato $\Leftarrow$
clausola $h \leftarrow a_1 \wedge \cdots \wedge a_n.$	atomo $h \Leftarrow a_1 \& \cdots \& a_n$
clausola $h.$	atomo $h \Leftarrow true$

- simboli $\&$ e $\Leftarrow$ *arbitrari*

Meta-Interprete Basilare

Meta-interprete basilare (*vanilla*) per il linguaggio delle clausole definite scritto nel medesimo linguaggio

- esteso nel seguito con altri costrutti e strumenti ingegneristici

Assiomatizzazione della relazione $prove$

- sue clausole *vere* nell'interpretazione intesa:
  - il meta-interprete copre ogni possibile query / corpo di clausola, specificando (anche in forma ricorsiva) un tipo di dimostrazione
- argomento = rappresentazione (meta) di un *goal* / *corpo* (base):
  - *vuoto*: $true$, prova immediata
  - *congiunzione* $A\&B$: provare sia A sia B
  - *atomo* H: provare una clausola con testa H e corpo B

Meta-Interprete *Vanilla*:

% $prove(G)$ vero se il corpo G (livello base) segue logicamente dalle clausole

- $prove(true).$
- $prove((A \& B)) \leftarrow prove(A) \wedge prove(B).$
- $prove(H) \leftarrow (H \Leftarrow B) \wedge prove(B).$

Esempio — Adattamento di KB precedente (atomi/fatti al meta-livello):

- $lit(L) \Leftarrow light(L)\ \&\ ok(L)\ \&\ live(L).$
- $live(W) \Leftarrow connected\_to(W, W_1)\ \&\ live(W_1).$
- $live(outside) \Leftarrow true.$
- $light(l_1) \Leftarrow true.$
- $light(l_2) \Leftarrow true.$
- $down(s_1) \Leftarrow true.$
- $up(s_2) \Leftarrow true.$
- $up(s_3) \Leftarrow true.$
- $connected\_to(l_1, w_0) \Leftarrow true.$
- $connected\_to(w_0, w_1) \Leftarrow up(s_2)\ \&\ ok(s_2).$
- $connected\_to(w_0, w_2) \Leftarrow down(s_2)\&\ ok(s_2).$
- $connected\_to(w_1, w_3) \Leftarrow up(s_1)\ \&\ ok(s_1).$
- $connected\_to(w_2, w_3) \Leftarrow down(s_1)\ \&\ ok(s_1).$
- $connected\_to(l_2, w_4) \Leftarrow true.$
- $connected\_to(w_4, w_3) \Leftarrow up(s_3)\ \&\ ok(s_3).$
- $connected\_to(p_1, w_3) \Leftarrow true.$
- $connected\_to(w_3, w_5) \Leftarrow ok(cb_1).$
- $connected\_to(p_2, w_6) \Leftarrow true.$
- $connected\_to(w_6, w_5) \Leftarrow ok(cb_2).$
- $connected\_to(w_5, outside) \Leftarrow true.$
- $ok(X) \Leftarrow true.$

(cont.)

- dato il goal (base) $live(w_5)$, query per meta-interprete:
  ask $prove(live(w_5))$.
  - si applica la terza clausola $prove$:
    - cercando nella KB una clausola (un atomo) $live(w_5) \Leftarrow B$ si trova $live(W) \Leftarrow connected\_to(W, W_1)\ \&\ live(W_1)$.
    - W e B associati, risp., a w_5 e $connected\_to(w_5, W_1)\ \&\ live(W_1)$.
  - si passa quindi a $prove((connected\_to(w_5, W_1)\ \&\ live(W_1)))$.
    - applicando $prove$, seconda clausola (per $\&$), 2 atomi da provare
  - ***primo*** atomo: $prove(connected\_to(w_5, W_1))$.
    - per la terza clausola di $prove$, si trova la clausola con testa unificabile $connected\_to(w_5, outside) \Leftarrow true$, per cui si associa W_1 a $outside$
      - $prove(true)$ immediato con la prima clausola di $prove$
  - ***secondo*** atomo: $prove(live(W_1))$
    - si riduce a $prove(true)$ immediatamente vero, essendo $W_1 = outside$

Estensioni del Linguaggio-Base

Il linguaggio-base può essere ampliato/ristretto modificando il meta-interprete:

- insieme di conseguenze dimostrabili *esteso*
  - aggiungendo clausole al meta-interprete
- insieme di conseguenze dimostrabili *ridotto*
  - aggiungendo condizioni alle clausole del meta-interprete

Estensione: Predicati Predefiniti

Nella pratica non tutti i predicati devono essere definiti da clausole:

- ad es. ingenuo assiomatizzare l'aritmetica: calcoli diretti più veloci
  - e.g. `sort(L1,L2)` ordinamento, `is(E1,E2)` unificazione, ...

Chiamata al *sottosistema* $call(G)$ valuta G *direttamente*:

- ad es. $call(p(X))$ equivale a provare $p(X)$
  - ma, nel linguaggio delle clausole, G (variabile) non sostituibile con atomi

Predicati predefiniti per procedure di livello-base:

- $built\_in(X)$ relazione (meta-livello)
  vera se tutte le istanze di X si possono valutare direttamente
  - variabile di meta-livello X denota un *atomo* di livello-base
  - a sua volta, $built\_in$ non necessariamente predefinito
    - può essere assiomatizzato come ogni altro predicato

Estensione: Disgiunzione

Disgiunzione $A \vee B$ ammessa nel corpo di una clausola:

- vera in un'interpretazione I quando almeno uno fra A e B è vero in I
  - NON si richiede la disgiunzione anche nel meta-linguaggio

Meta-interprete Esteso

Meta-interprete per un linguaggio che ammette le due estensioni:

% $prove(G)$ vero se il corpo (base) G è conseguenza logica della KB (base)

- $prove(true).$
- $prove((A \& B)) \leftarrow prove(A) \wedge prove(B).$
- $prove((A \vee B)) \leftarrow prove(A).$
- $prove((A \vee B)) \leftarrow prove(B).$
- $prove(H) \leftarrow built\_in(H) \wedge call(H).$
- $prove(H) \leftarrow (H \Leftarrow B) \wedge prove(B).$

- dato un DB di asserzioni predefinite
- si assume che $call(G)$ sia un modo per dimostrare G al meta-livello

Esempio — regole a livello-base interpretabili dal meta-interprete:

- $can\_see \Leftarrow eyes\_open \;\&\; (lit(l_1) \vee lit(l_2))$.
  - can_see vero se veri $eyes\_open$ e almeno uno tra $lit(l_1)$ e $lit(l_2)$

Con questo meta-interprete, meta-linguaggio diverso dal linguaggio-base:

- *linguaggio-base*: ammette la disgiunzione nel corpo
- *meta-linguaggio*: non richiede la disgiunzione per fornirla al linguaggio-base
  - ma richiede un modo per interpretare $call(G)$
    non gestibile a livello base
  - per interpretare $call(G)$ a livello-base
    si può aggiungere una clausola di meta-livello

$$prove(call(G)) \leftarrow prove(G).$$

Ragionamento a Profondità Limitata

Per *specializzare* il ragionamento: aggiungendo *condizioni* a clausole di meta-livello si può *limitare* quello che può essere dimostrato

Ragionamento a profondità limitata tramite meta-interprete:

- si cercano dimostrazioni *brevi*
  come parte di uno schema di *iterative deepening*:
  - ripetere DFS limitate con limiti di profondità crescenti

% $bprove(G, D)$ vero se G dimostrabile con dimostrazione di profondità massima D

- $bprove(true, D).$
- $bprove((A \& B), D) \leftarrow bprove(A, D) \wedge bprove(B, D).$
- $bprove(H, D) \leftarrow D \geq 0 \wedge D_1 \; is \; D - 1 \wedge (H \Leftarrow B) \wedge bprove(B, D_1).$

Osservazioni

- usato il predicato infisso is (come in PROLOG):
  - $V\ is\ E$ vero se V valore (numerico) dell'espressione E
  - in E, "$-$" indica la sottrazione (funzione infissa)
    - ad es. $V\ is\ D - 1$ vero se il valore di V si ha sottraendo 1 a quello di D
- se D è limitato da un numero nella query, non può divergere
  - non tratterà dimostrazioni di lunghezza maggiore → interprete *incompleto*
- ogni dimostrazione trovata dal meta-interprete $prove$ può essere trovata da $bprove$ se il valore D è sufficientemente grande
  - idea dell'iterative-deepening
- $bprove$ potrebbe trovare dimostrazioni che $prove$ non riesce a trovare
  - $prove$ può andare in loop prima d'aver esplorato tutte le dimostrazioni

Altri Meta-interpreti Limitati

Alternative:

- realizzati usando una diversa *misura* sugli alberi di dimostrazione
- esempi:
  - *numero di nodi* nell'albero
  - assumendo un costo sulle congiunzioni e modificando la seconda regola di $bprove$

Meta-Interprete per Alberi di Dimostrazione

Per implementare le domande how, meta-interprete che costruisce esplicitamente l'albero di dimostrazione per una risposta:

- ammette predicati built-in e costruisce una rappresentazione dell'albero
  - da attraversare per le rispose alle domande how
  - un albero potrà essere:
    - a singolo nodo (atomo): $true$ o $built\_in$
    - $if(G, T)$ con G atomo e T albero
    - $(L\&R)$ con L e R sotto-alberi

% $hprove(G, T)$ vero se il corpo G (l.base) è conseguenza logica della KB (l.base) con T albero di dimostrazione

- $hprove(true, true).$
- $hprove((A\&B), (L\&R)) \leftarrow hprove(A, L) \wedge hprove(B, R).$
- $hprove(H, if(H, built\_in)) \leftarrow built\_in(H) \wedge call(H).$
- $hprove(H, if(H, T)) \leftarrow (H \Leftarrow B) \wedge hprove(B, T).$

Esempio — KB sulla domotica e query ask $lit(L)$ (l. base)

- sola risposta: $L = l_2$
- query a meta-livello: ask $hprove(lit(L), T)$
  - albero risultante T:

$$
\begin{aligned}
T = if(&lit(l_2),\\
&if(light(l_2), true)\ \&\\
&if(ok(l_2), true)\ \&\\
&if(live(l_2),\\
&\quad if(connected\_to(l_2, w_4), true)\ \&\\
&\quad if(live(w_4),\\
&\quad\quad if(connected\_to(w_4, w_3),\\
&\quad\quad\quad if(up(s_3), true))\ \&\\
&\quad\quad if(live(w_3),\\
&\quad\quad\quad if(connected\_to(w_3, w_5),\\
&\quad\quad\quad\quad if(ok(cb_1), true))\ \&\\
&\quad\quad\quad if(live(w_5),\\
&\quad\quad\quad\quad if(connected\_to(w_5, outside), true)\ \&\\
&\quad\quad\quad\quad if(live(outside), true))))))
\end{aligned}
$$

  - le domande how richiedono il suo attraversamento per mostrare all'utente le clausole usate (non l'intero albero)

Differire i Goal

Capacità utili nei meta-interpreti: **differire** gli obiettivi

- mettere atomi in lista d'attesa invece di dimostrarli subito
- alla fine della dimostrazione, si può derivare l'implicazione:
  se i goal differiti sono tutti veri allora la risposta calcolata è giusta

Utilizzi: nelle implementazioni di

- *dimostrazione per contraddizione*: e.g. per la CBD
- *ragionamento abduttivo*: differiti gli assumibili
- dimostrazioni di atomi *con variabili*: in attesa del loro *grounding*
- *regole che evitino passi intermedi*
  - es. goal differiti da provare interrogando l'utente o un DB

% $dprove(G, D_0, D)$ vero se G segue logicamente dalla congiunzione degli atomi differibili in D, con D_0 sottolista di D

- $dprove(true, D, D).$
- $dprove((A\&B), D_1, D_3) \leftarrow dprove(A, D_1, D_2) \wedge dprove(B, D_2, D_3).$
- $dprove(G, D, [G|D]) \leftarrow delay(G).$
- $dprove(H, D_1, D_2) \leftarrow (H \Leftarrow B) \wedge dprove(B, D_1, D_2).$

Osservazioni

- atomo di l.base G reso ***differibile*** attraverso il fatto (meta-l.) $delay(G)$
  - va in una lista: bypassando la dimostrazione
- dimostrato $dprove(G, [\,], D)$, allora $G \Leftarrow D$ conseguenza logica, con $delay(d)$ vero per ogni $d \in D$
  - nuova clausola $G \Leftarrow D$ istanza di **valutazione parziale**
    - alla base dell'**explanation-based learning** che tratta le clausole derivate come clausole apprese che possono sostituire quelle originarie

Esempio — (CBD) Considerata la KB sulla domotica, si escludano le regole per ok, predicato che può essere reso ***differibile*** aggiungendo il fatto: $delay(ok(G))$.

- query: ask $dprove(live(p_1), [\,], D)$.
- risposta: $D = [ok(cb_1)]$
  - i.e. se $ok(cb_1)$ fosse vero allora lo sarebbe $live(p_1)$
- query: ask $dprove((lit(l_2)\ \&\ live(p_1)), [\,], D)$.
- risposta: $D = [ok(cb_1), ok(cb_1), ok(s_3)]$
  - i.e. se cb_1 e s_3 sono ok allora l_2 sarà lit e p_1 sarà $live$
  - si noti che $ok(cb_1)$ compare due volte nella lista
    - $dprove$ non controlla occorrenze multiple di differibili nella lista: facile modifica

Condivisione della Conoscenza

Interoperabilità

Prima della rappresentazione,
acquisizione di conoscenza da persone e dati:

- conoscenza, per ogni dominio proviene
  - da diverse sorgenti
  - in momenti diversi
- **interoperabilità** tra sorgenti diverse:
  capacità di lavorare insieme a livello *sintattico* e *semantico*

ONTOLOGIA

Specifica del *significato* dei simboli in un *KBS*:

- base di conoscenza, sistema di sensori (ad es. termometri), altre tipologie...

Significato spesso può risiedere (informalmente):

- nella mente del progettista
- in un manuale-utente
- nei commenti aggiunti alle KB

Specifica *formale* per l'**interoperabilità semantica**:

- abilità da parte di diverse KB di collaborare a livello semantico, nel rispetto del significato dei simboli

Esempio — Un web-bot per acquisti trova su un sito info su un prodotto denominato `chips` e deve capire se si tratti di *patatine*, *parti di computer*, *pezzi di legno*, *gettoni per il gioco*

- specificando il significato della *terminologia* adottata mediante un'ontologia:
  - preferire a `chip` il simbolo `WoodChipMixed` definito nell'ontologia di un'organizzazione autorevole
  - la rappresentazione formale dovrebbe usare `WoodChipMixed` avendone dichiarato l'adozione per *evitare ambiguità*
  - terze parti potrebbero dichiarare che `ChipOfWood` usato in un'altra KB corrisponde a `WoodChipMixed` di questa, con un *mapping* esplicito

Dettaglio e Flessibilità

- Specifica *non* necessariamente *dettagliata*
  ma atta almeno a garantirne la *coerenza* nell'uso
  - ad es. utile specificare l'unità di misura di un termometro
    - anche senza definire cosa sia una temperatura o la sua accuratezza
- Uso della *logica* → rappresentazioni flessibili:
  - aggiunta *modulare* di conoscenza
    - anche estensione delle relazioni con argomenti aggiuntivi
- Specifica del significato usabile per
  - l'*acquisizione* di nuova conoscenza
  - la *spiegazione*
  - il *debugging* a livello di conoscenza

Rappresentazioni Flessibili

Rappresentazioni

Alla base delle moderne *ontologie* rappresentazioni flessibili supportate da strumenti logici:

- *Individui e relazioni*
- Rappresentazioni *grafiche*
- *Classi*

Scegliere Individui e Relazioni

Dato un *linguaggio* di rappresentazione logico e un *mondo* sul quale ragionare, i progettisti di devono scegliere *individui* e *relazioni*:

- decidere come decomporre il mondo d'interesse sta a chi lo modella
  - livello di dettaglio dipendente dal compito da svolgere

Esempio — `red` proprietà ascrivibile a oggetti del mondo reale, in base allo *spettro di frequenze* della luce assorbite o riflesse

- indicando un certo insieme/intervallo di frequenze
- preferendo una mappatura su termini: `pink`, `scarlet`, `ruby` e `crimson`
- dividendolo in regioni non corrispondenti a termini del linguaggio ma più utili a distinguere diverse categorie di individui

Relazioni da definire seguendo alcune *linee guida* ingegneristiche

Esempio — red categoria di classificazione appropriata per gli individui

- come *relazione unaria*:
  - l'oggetto a è rosso → $red(a)$
  - query possibile: *"Cosa si conosce di colore rosso?"* → ask $red(X)$
    - valori di X restituiti: individui rossi
  - difficile chiedere *"Di che colore è l'oggetto a ?"*
    - nel linguaggio delle clausole definite, non si può chiedere ask $X(a)$
      - i nomi dei predicati non possono essere variabili
      - logiche di ordine superiore: possibile per qualsiasi proprietà di a

- alternativa: colore come *individuo*
  - costante red per denotare il rosso
  - predicato $color(Ind, Val)$: Val colore dell'individuo Ind
    - es. *"l'oggetto a è rosso"* → $color(a, red).$
  - colori come individui nominabili come altri oggetti (es. plico)
    - $color$ relazione binaria tra individui fisici e colori
  - ora si può chiedere *"Cosa è di colore rosso?"* con
    ask $color(X, red)$
    e *"Di che colore è l'oggetto a ?"* con
    ask $color(a, C).$

Riduzione di un concetto astratto a oggetto: forma di **reificazione**

Esempio — Ulteriori estensioni

- $color$ come predicato non consente domande come
  - *"Quale proprietà di questo oggetto ha valore red?"*
    - risposta: $color$
- trasformazione: *proprietà come individuo* con la nuova relazione $prop$
  - es. *"l'individuo a ha per la proprietà $color$ il valore red"*
    → $prop(a, color, red)$
  - possibili tutte le query viste prima
    riscrivendo tutte le relazioni in termini di $prop$

Rappresentazione individuo–proprietà–valore

$$prop(Ind, Prop, Val)$$

- l'individuo Ind ha per la proprietà $Prop$ il valore Val

Tripla di elementi `<s,p,o>`

- `s`: **soggetto**
- `p`: **predicato verbale**
- `o`: **oggetto**

scritta anche come ***enunciato***:

$$soggetto \quad verbo \quad oggetto.$$

equivalente all'***atomo***

$$prop(soggetto, verbo, oggetto)$$

o, in notazione ***funzionale***:

$$verbo(soggetto, oggetto)$$

In una tripla, il *verbo* è una proprietà p con

dominio insieme di individui che possono essere soggetti in triple con verbo p

codominio insieme di valori che possono essere oggetti in triple con verbo p

Attributo: coppia *proprietà-valore*

- ad es. un certo plico ha *colore rosso*

Esempio — Per trasformare in forma di tripla $parcel(a)$:

1. $parcel$ come concetto (reificato):

$$prop(a, type, parcel).$$

   - l'individuo a è nella classe $parcel$
   - $type$ proprietà speciale che collega individui a classi
     - spesso indicato con `is_a` (relazione $\in$ della matematica)
   - costante $parcel$ denota la ***classe***:
     - insieme di tutte le cose, reali o potenziali, che sono plichi

2. $parcel$ come proprietà ***booleana***, vale $true$ per individui che sono plichi:

$$prop(a, parcel, true).$$

   - una ***proprietà booleana*** ha codominio $\{true, false\}$ con $true$ e $false$ costanti

Esempio — Predicati complessi in forma di *triple*: es. CLASSBOOK

- relazione: $scheduled(C, S, T, R)$ prenotazione corsi/aule
  - nel corso C, modulo/sezione S *programmata* per l'ora T in aula R
    - es. $scheduled(cs422, 2, 1030, cc208).$ formalizza
      "*lezione sulla sezione 2 del corso cs422 prevista per le 10:30 in aula cc208*"
- rappresentazione in triple: $scheduled$ reificata
  - individuo-prenotazione (*booking*) con un nome e le proprietà:
    - il corso: $course$
    - la parte del corso: $section$
    - l'orario di inizio: $start\_time$
    - l'aula: $room$
  - $scheduled(cs422, 2, 1030, cc208).$ → oggetto $b123$ nelle triple:
    - $prop(b123, course, cs422).$
    - $prop(b123, section, 2).$
    - $prop(b123, start\_time, 1030).$
    - $prop(b123, room, cc208).$

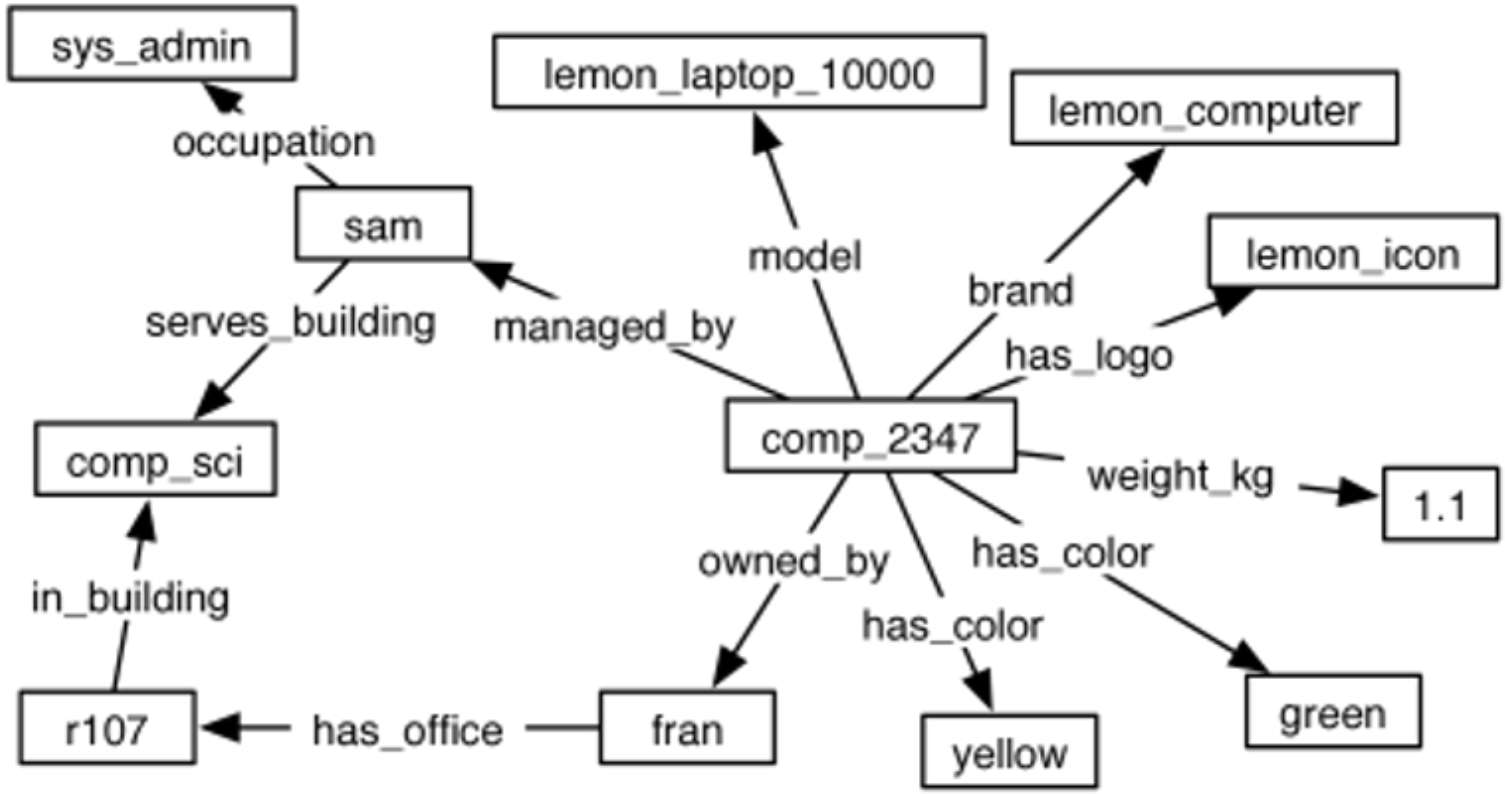
(..cont.)

- vantaggi della rappresentazione in triple: *modularità*
  - evidenzia i *valori appropriati* per ogni proprietà
  - facile *aggiungere* proprietà, come il docente o la durata
    - ad es.
      "*Fran terrà la lezione sulla sezione 2 del corso cs422 programmata per le 10:30 in aula cc208 della durata prevista di 50*"
      aggiungendo:
      - $prop(b123, instructor, fran).$
      - $prop(b123, duration, 50).$
    - difficile usando il predicato $scheduled$
      - andrebbe esteso il numero di argomenti

Relazione $prop$ interpretabile in termini di *grafo orientato*:

- $prop(Ind, Prop, Val)$ raffigurato con
  - ***nodi***: Ind e Val
  - ***arco***: etichettato da $Prop$ tra di essi
- grafo risultante: **rete semantica** o *knowledge graph* [11]
  - mappatura diretta su una base di conoscenza tramite la relazione $prop$

Esempio — Rete semantica su (consegne PC in) un dominio universitario



- alcune relazioni rappresentate:
  - $prop(comp\_2347, owned\_by, fran).$
  - $prop(comp\_2347, managed\_by, sam).$
  - $prop(comp\_2347, model, lemon\_laptop\_10000).$
  - $prop(comp\_2347, brand, lemon\_computer).$
  - $prop(comp\_2347, has\_logo, lemon\_icon).$
  - $prop(comp\_2347, color, green).$
  - $prop(comp\_2347, color, yellow).$
  - $prop(comp\_2347, weight, light).$
  - $prop(fran, has\_office, r107).$
  - $prop(r107, in\_building, comp\_sci).$
- mostra anche la strutturazione della conoscenza:
  - ad es. facile comprendere che il computer `2347` appartiene a qualcuno (`Fran`) il cui ufficio (`r107`) è nell'edificio di Informatica (`comp_sci`)

Vantaggi della notazione grafica:

- facile per gli umani visualizzare relazioni
  senza dover imparare la sintassi di un dato linguaggio logico
  - ausilio per i progettisti nell'organizzazione della conoscenza
- si possono anche ignorare le etichette con nomi privi di significato
  - `comp_2347` in figura o `b123` usato prima
  - ammissibili nodi **blank**, privi di etichetta
    - serve dare un nome arbitrario solo in caso di mappatura in forma logica

Tipicamente, di un dominio si conoscono e formalizzano:

- database di *fatti*
- *regole* generali dalle quali derivare altri fatti
  - modalità ⟶ scelta progettuale

Conoscenza Primitiva asserita/specificata esplicitamente in termini di *fatti*

Conoscenza Derivata inferita da altra conoscenza e specificata attraverso *regole*

REGOLE

Rappresentazione più compatta:

- relazioni derivate: per trarre conclusioni a partire da osservazioni
  - non tutto è osservabile!
    - conoscenza in gran parte *inferita* da osservazioni e
      da conoscenza più generale disponibile
- usi standard:
  - raggruppare gli individui in classi
  - associare *proprietà* generali alle classi in modo che i loro individui le *ereditino*
    → rappresentazioni *concise*
    - i membri di una classe condividono attributi comuni
      - cfr. classificatori probabilistici (NB)

Classe

insieme di individui membri *effettivi* e *potenziali*, definita in forma

- *intensionale* — tramite **funzione caratteristica**
  - `1`/`true` per i membri dell'insieme e `0`/`false` per gli altri individui
- *estensionale* — elenco degli elementi
  - ad es. classe `Chair` insieme di tutte le cose che possono essere *sedie*
    - non limitato a quelle già osservate,
      per non escludere sedie ancora da produrre

equivalenza tra classi non limitata ai soli membri conosciuti

- ad es. la classe degli unicorni verdi e quella delle sedie alte 100m, potrebbero contenere gli stessi elementi (nessuno) eppure essere diverse

Classi come Tipi Naturali

La definizione consente di descrivere *qualsiasi* insieme come classe:

- es. classe come insieme che include:
  - il numero `18`,
  - la `Curva_Nord_S_Nicola_di_Bari`,
  - il `piede_sinistro_di_A_Cassano` considerato come classe

utile ?

Tipo naturale: insieme/classe che rende la descrizione più concisa

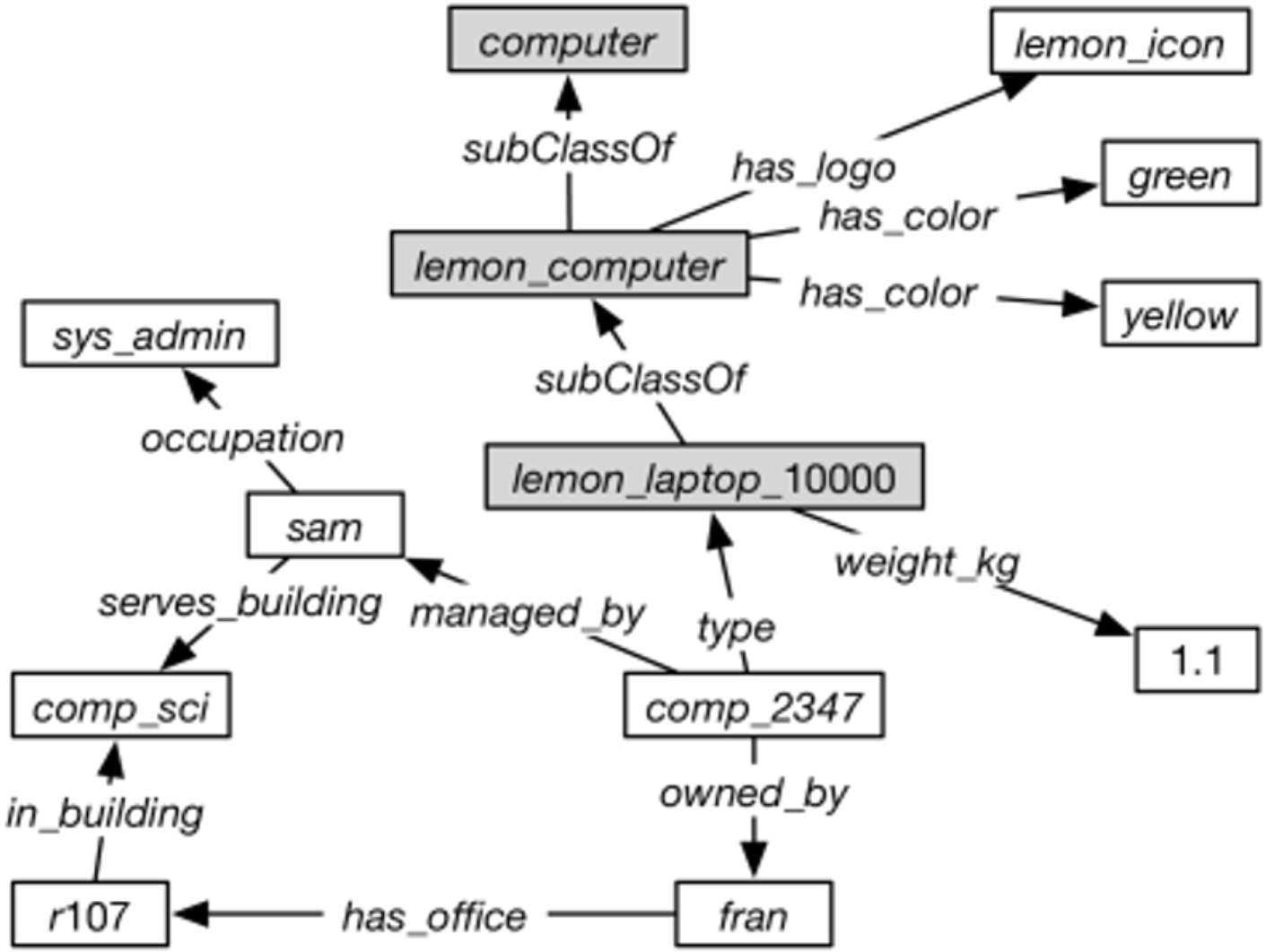
- ad es. `mammifero` tipo naturale
  - descrive attributi comuni ai mammiferi
  - rende più compatta la KB, evitando ripetizioni per ogni singolo individuo

Sottoclassi

S sottoclasse di C, relazione di ***sottoinsieme***:
ogni individuo di tipo S è di tipo C anche quelli futuri!

Esempio — L'esempio precedente specificava esplicitamente che il logo del computer $comp\_2347$ era l'icona di un limone

- sapendo che vale per tutti i computer Lemon, può essere associato a $lemon\_computer$ e poi derivato per il particolare $comp\_2347$
- ***vantaggio*** — inferenza valida per tutti i computer di quella marca
  - analogamente si può codificare che ogni $lemon\_laptop\_10000$ pesi 1.1kg
- estensione del caso precedente in ***figura*** ↻
  - rettangoli più scuri: ***classi***
  - archi dalle classi: proprietà di ciascun membro della classe
    - es. $comp\_2347$ pesa 1.1kg, non l'insieme dei $lemon\_laptop\_10000$



Rete semantica con ereditarietà. Nodi-classe più scuri

Classi ed Ereditarietà

Relazione tra tipi e sottoclassi definibile come *clausola*:

$$prop(X, type, C) \leftarrow prop(S, subClassOf, C) \wedge prop(X, type, S).$$

Proprietà speciali $type$ e $subClassOf$
per specificare l'**ereditarietà di proprietà**:

- valori di proprietà, specificati a livello di classe, sono ereditati dalle sue istanze
- "*tutti i membri di classe* c *hanno il valore* v *per la proprietà* p", in DATALOG:

$$prop(Ind, p, v) \leftarrow prop(Ind, type, c).$$

  - insieme alla regola precedente che collega tipi e sottoclassi,
    serve a far ereditare (valori di) proprietà

Esempio — Tutti i computer Lemon hanno un limone come logo e i colori giallo e verde (cfr. figura precedente)

- con un programma DATALOG:
  - $prop(X, has\_logo, lemon\_icon) \leftarrow prop(X, type, lemon\_computer).$
  - $prop(X, has\_color, green) \leftarrow prop(X, type, lemon\_computer).$
  - $prop(X, has\_color, yellow) \leftarrow prop(X, type, lemon\_computer).$
  - $prop(X, weight\_kg, 1.1) \leftarrow prop(X, type, lemon\_laptop\_10000).$
  - $prop(lemon\_laptop\_10000, subClassOf, lemon\_computer).$
  - $prop(comp\_2347, type, lemon\_laptop\_10000).$
- da tale programma e dalla clausola su $subClassOf$ vista prima si possono derivare info su logo, colori e peso di $comp\_2347$
- per includere un nuovo computer Lemon Laptop 10000
  - dichiarare che è un Lemon Laptop 10000: colori, logo e peso derivati per ereditarietà

per decidere tra conoscenza primitiva e derivata

LINEE GUIDA

- per *associare un attributo* a un individuo i, lo si associa alla classe C più generale cui i appartenga, tale che tutte le sue istanze abbiano l'attributo
  - l'ereditarietà fa associare l'attributo all'individuo e alle altre istanze di C
  - metodologia che tende a rendere le KB più concise:
    - più facile incorporare nuovi individui: erediteranno l'attributo
- non associare a una classe un **attributo contingente**
  - i.e. il cui valore cambi con le circostanze
    - ad es. potrebbe essere vero nel contesto corrente che tutti i computer sono venduti in scatole di cartone
    - potrebbe non essere una buona idea definirlo come attributo della classe computer perché potrebbe non valere per computer acquistati in futuro
- assiomatizzare nella *direzione causale*:
  - se esiste una scelta tra rendere primitiva la causa o l'effetto, si renda primitiva la causa, verosimilmente più stabile al variare del dominio

Classi nelle Basi di Conoscenza e in OOP

Individui e classi *comuni* a KB in logica e OOP con alcune *differenze*:

- Nell'OOP oggetti *computazionali*:
  - strutture dati e programmi associati
    - `persona` oggetto in Java, non è una persona
  - in una KB (tipicamente) cose del mondo reale
    - `persona` individuo in una KB che corrisponde a un essere reale
  - ad es. `sedia`
    - sedia reale su cui ci si può sedere, può far male urtandola
      - tipicamente in una KB non ci si interagisce ma ci si ragiona
      - ferma a meno di non essere spostata da un agente fisico
    - a una sedia-oggetto si può mandare un messaggio e ottenere risposta

- In una KB, rappresentazione di un oggetto come *approssimazione* a uno o più livelli di astrazione
  - oggetti reali più complessi
    - non si rappresentano individui come le fibre nel suo legno
  - In OOP, esistono solo le proprietà rappresentate di un oggetto
    - il sistema può sapere tutto su un oggetto, ma non dell'individuo reale corrispondente
- Struttura di una classe in OOP tesa a rappresentare il *progetto* di oggetti
  - lavoro dell'analista-programmatore
    - ad es., in Java, oggetto membro di una sola classe a livello minimo: no ereditarietà multipla
  - oggetti reali non si comportano sempre in modo preciso e prevedibile
    - una stessa persona potrebbe essere portiere di calcio, medico e padre

- Un programma non ammette *incertezza* riguardo le sue strutture dati
  - deve selezionare le strutture utili
  - del mondo reale invece si può assumere incertezza sui tipi degli individui
- Le rappresentazioni degli individui nelle KB non svolgono *azioni*
  - in una KB, rappresentano soltanto oggetti del dominio cui si riferiscono
  - in OOP, gli oggetti svolgono lavoro computazionale
- Si può usare un linguaggio di *modellazione OO*, come UML, per rappresentare una KB, ma non è la scelta migliore:
  - nella modellazione OO, facility per costruire (buoni) progetti mentre il mondo reale da modellare potrebbe non essere basato su un buono schema
  - forzare un buon paradigma di progettazione su un dominio confuso potrebbe non essere produttivo

Ontologie e Condivisione della Conoscenza

Complessità della costruzione di grandi sistemi basati su conoscenza:

- conoscenza spesso da *più sorgenti*: va integrata
  - possono non condividere lo stesso modo per analizzare il mondo
  - semplificano il mondo secondo specifiche priorità:
    - campi diversi con terminologie peculiari
- i sistemi *evolvono* nel tempo:
  - difficile anticipare distinzioni utili in futuro
- mondo spesso *indistinto*
  - i progettisti devono scegliere *individui e relazioni* da rappresentare
  - devono accordarsi su una decomposizione conveniente del dominio
- difficile tenere a mente *notazione* e *significato* propria e altrui:
  - dato un simbolo utilizzato, determinarne il significato
  - dato un concetto, determinare con che simbolo rappresentarlo:
    - concetto già definito?
    - associato a quale simbolo?
    - altrimenti, quali concetti correlati con cui definirlo?

Concettualizzazione

Per condividere e comunicare conoscenza, importante sviluppare un *vocabolario* con un suo *significato* comunemente accettato

Concettualizzazione: associazione tra vocabolario di simboli usati nella macchina e individui / relazioni del mondo

- particolare *astrazione* del mondo e la sua *notazione*
  - *in piccolo*: documentazione curata dal progettista
    - spesso informale
    - poco scalabile
  - *grandi sistemi*: concettualizzazione *condivisa*

Ontologia vs. ontologia

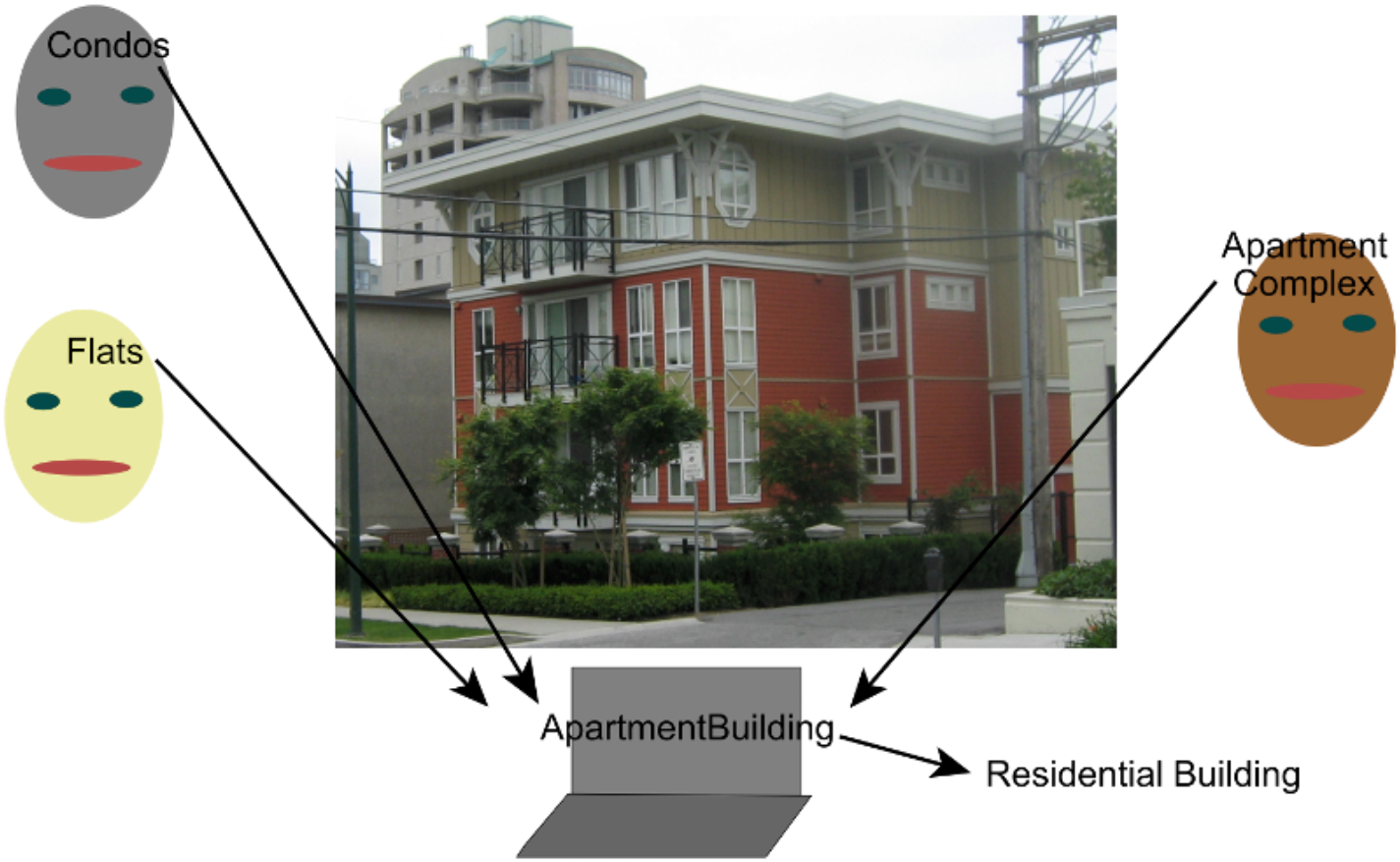
Ontologia (*Filosofia*) — studio dell'esistente

ontologia (*Informatica/AI*) — specifica formale dei significati dei simboli manipolati da un KBS:

- *concettualizzazione* di individui e relazioni assunti come esistenti
- *terminologia* con:
  - tipi di individui da modellare
  - proprietà da usare
  - assiomi che vincolano l'uso del vocabolario

Esempio — Un'ontologia riguardante le *mappe* potrebbe specificare che `ApartmentBuilding` rappresenti edifici con appartamenti:

- non definirà un edificio, ma lo descriverà in modo che gli *altri* ne comprendano la definizione
- altri, che potrebbero chiamarli `Condos`, `Flats` o `Apartment Complex`, dovrebbero poter trovare il simbolo appropriato nell'ontologia
  - associazioni: concetto → simbolo → significato



Associazione dalla concettualizzazione a un simbolo

Un'ontologia può contenere *assiomi* per vincolare l'uso dei simboli

Esempio — (cont.)

- specificando che gli `Apartment Building` siano `Building`,
  quindi *opere* costruite da uomini
- imponendo qualche restrizione sulle dimensioni degli edifici
  in modo da escludere, ad es., `Box` o intere `City`
- dichiarando che un edificio non possa trovarsi simultaneamente in locazioni geograficamente distanti
  - se si staccasse una sua parte spostandola in un posto diverso,
    non sarebbe più un edificio singolo
  - essendo un `Apartment Building` anche `Building`,
    le restrizioni si applicano anche ad ognuno di essi

Un'**ontologia** è composta da:

- un *vocabolario* delle categorie di cose da rappresentare
  - classi e proprietà
- un'*organizzazione* delle categorie
  - ad es. gerarchia d'ereditarietà attraverso proprietà speciali come `subClassOf` o `subPropertyOf`, o altre modalità
- un insieme di *assiomi* che vincolano la definizione di alcuni simboli per riflettere meglio il significato inteso
  - ad es. transitività d'una proprietà, restrizioni su dominio, codominio, o sul numero di valori che la proprietà può assumere per individuo
  - a volte relazioni definite in termini di più relazioni primitive, i.e. senza definizione (intensionale)

Non serve specificare individui *sconosciuti* nella progettazione, solo individui prefissati da condividere:

- ad es. giorni della settimana o colori

Osservazioni

- Le ontologie sono spesso definite *indipendentemente* dalle applicazioni ma richiedono l'*accordo* di una comunità sul significato dei simboli
- Un'ontologia specifica il significato dei simboli per gli utenti consentendo l'*interoperabilità* fra diverse KB
  - fornisce il *collante semantico* per unire richieste degli utenti e KB

Esempio — dominio immobiliare:

- Gli utenti descrivono la *sistemazione desiderata*
- Il sistema potrà
  - *cercare* su più KB le sistemazioni più consone
  - *contattare* gli utenti quando si libera una sistemazione appropriata
- Considerare case indipendenti e condomini come edifici residenziali:
  - utile per suggerire di affittare una casa / un appartamento,
    ma non un intero condominio (edificio)
  - concetto di *unità abitativa* / "living unit" come gruppo di stanze in cui si convive
    (tipica offerta delle agenzie)
    - singola stanza, o persino una parte / un posto letto
  - casi limite, non previsti in partenza, o spesso non definiti chiaramente
    ma delineati in seguito nel corso dell'evoluzione dell'ontologia
- Solo conoscenza (più) stabile e generale
  - descrizioni/offerte reali: ignoti in fase di progettazione

Usi delle Ontologie

Scopo primario: *documentare* il significato dei simboli,
associare simboli (nella macchina) a concetti (nella mente del progettista)

- dato un simbolo, usare l'ontologia per determinare cosa significhi
- dovendo rappresentare un concetto, si sfrutta l'ontologia per trovare il simbolo appropriato o per evidenziare la necessità di una nuova definizione

Scopo secondario: usando gli assiomi, consentire l'*inferenza*
o individuare le eventuali *contraddizioni*

- problema principale nella progettazione: *organizzazione* dei concetti (simboli) in modo che la macchina possa inferire conoscenza utile dai fatti asseriti

Interoperabilità

Sintattica

XML (*Extensible Markup Language*) linguaggio che fornisce una *sintassi* progettata per l'elaborazione automatica ma leggibile da persone:

- linguaggio testuale, con elementi costituiti da *tag* organizzati gerarchicamente
- sintassi anche complessa, ma al livello più semplice l'ambito di un tag è della forma `<tag.../>` oppure `<tag...>` … `</tag>`

Semantica:

- chiunque può costruire un'ontologia
- nel progettare e realizzare una KB si possono usare ontologie esistenti o svilupparne altre sulla base di quelle esistenti
- per l'*interoperabilità semantica*, aziende e singoli dovrebbero tendere a
  - riusare ontologie *standard* per i dominio interessati
  - e/o definire *mapping* dalle proprie verso ontologie standard
    - cfr. tentativi per costruire grandi ontologie universali, come Cyc

Web Semantico

Web Semantico — prospettiva che prevede la distribuzione di *conoscenza* attraverso l'infrastruttura del Web (server):

- ~~doc HTML~~ destinati alle persone
- conoscenza *interpretabile dalle macchine*: semantica esplicitata attraverso ontologie → ragionamento

Basi:

- URI
- RDF (SPARQL), RDF-Schema
- OWL

Uniform Resource Identifier

URI (*Uniform Resource Identifier*) identifica *univocamente* una *risorsa*:

- qualsiasi cosa possa essere identificata
  - compresi individui, classi e proprietà
- sintassi basata su quella degli URL: `<url#name>`
  - dove `url` è l'indirizzo di una pagina Web
    - con HTTP si può *negoziare* il tipo di risposta (HTML/RDF) in base alla richiesta del client
  - oggi anche IRI, con set-caratteri esteso
- esempi:
  - individuo `<http://example.org/#spiderman>`
  - proprietà `<http://xmlns.com/foaf/0.1/name>`
- *abbreviazione* (o *CURIE*) `abbr:name`
  - `abbr` *prefisso* dell'URI completo (*namespace*) dichiarato localmente
- ogni URI ha un significato convenzionale condiviso attraverso l'uso

Esempio — Ontologia **foaf** (*friend-of-a-friend*) per pubblicare informazioni su persone, reti di amici, ...

- URI associati al *namespace* `<http://xmlns.com/foaf/0.1>`
  - abbreviato con il prefisso `foaf`
- `foaf:name` proprietà che correla una persona a una rappresentazione del suo nome (una stringa)
  - consentirà di sapere esattamente a quale proprietà ci si riferisca
  - URI della prop. formato premettendo il prefisso del namespace
- `foaf:knows` per collegare una persona ai suoi amici
- ...

RDF — SPARQL — RDF-Schema

RDF (*Resource Description Framework*) *modello di dati* in forma di *triple*:

`individuo–proprietà–valore`

- sintassi (serializzazione) costruita su XML → RDF/XML
  - ma esistono altri formati più leggibili come Turtle, N3,...

```
URI_1 URI_2 Valore.
URI_1 URI_2 URI_3.
```

- esempio: `<http://example.org/#spiderman>`
  `<http://xmlns.com/foaf/0.1/name> "Uomo Ragno"@it .`

NB RDF ammette la *reificazione di enunciati*:

- formule logiche arbitrarie: quindi, in generale, <u>non è decidibile</u>
  - non sempre un problema:
    - significa solo non poter limitare il tempo necessario a un dato calcolo
  - idem per i programmi logici con funzioni (o i programmi in qualsiasi linguaggio)

SPARQL protocollo+linguaggio per l'interrogazione di KB, grafi di triple RDF:

- attraverso *endpoint*:
  - es. DBpedia, Min. Istruzione, INPS, Regione Puglia, ...
- query definite da *pattern* fatti da triple con *variabili*

Esempio — Testabile attraverso l'endpoint di DBpedia

```
select ?canzone ?data where {
  dbr:The_Sugarcubes dbo:formerBandMember ?componente .
  ?componente rdf:type yago:Female109619168 .
  ?canzone dbo:artist ?componente .
  ?canzone dbo:releaseDate ?data .
} limit 3
```

risultati:

```
http://dbpedia.org/resource/Cosmogony_(song) 2011-07-19
http://dbpedia.org/resource/Hunter_(Björk_song) 1998-10-05
http://dbpedia.org/resource/Declare_Independence 2008-01-01
```

RDF–S (*RDF Schema*)

consente di definire risorse, classi / proprietà, in termini di altre risorse

Esempio

```
@prefix : <http://www.example.org/sample.rdfs#>.
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
:Dog       rdfs:subClassOf :Animal.
:Person    rdfs:subClassOf :Animal.
:hasChild  rdfs:range :Animal;
           rdfs:domain :Animal.
:hasSon    rdfs:subPropertyOf :hasChild.

:Max       rdf:type   :Dog.
:Abel      a          :Person.
:Adam      :hasSon    :Abel.
```

- consente anche di *restringere* domini e range di proprietà
- fornisce anche *contenitori*: set, sequenze e alternative

I linguaggi ontologici come OWL si basano sulle **logiche descrittive** [DL]:

- servono a descrivere classi, proprietà e individui
- *idea* fondante: separazione della *base di conoscenza* $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$
  - $\mathcal{T}$ parte **terminologica** (**TBox**) che descrive la terminologia: *assiomi* per definire il significato dei simboli
    - es. $C \sqsubseteq (C_1 \sqcap \exists R_1 . C_2) \sqcup \forall R_2 . (C_3 \sqcap \neg C_4)$
    - definita in fase di progettazione del sistema, ne specifica l'ontologia
      - assiomi OWL (diverse sintassi)
      - stabile: a meno che non cambi il significato del vocabolario (caso raro)
  - $\mathcal{A}$ parte **asserzionale** (**ABox**) specifica verità fattuali
    - es. in DL: $C(a)$, $R(a, b)$, $P(b, v)$
    - conoscenza su situazioni contingenti (stato del mondo)
      - asserzioni RDF (triple)
      - conoscenza nota completamente solo al momento dell'utilizzo (*runtime*)

OWL

OWL (*Web Ontology Language*) linguaggio per ontologie su Web

Con OWL è possibile descrivere mondi/domini in termini di:

- **Individui** — *entità* del mondo che si descrive
  - e.g., URI di una data casa o una particolare prenotazione
- **Classi** — *insiemi di individui*
  - tutte le entità reali o potenziali che potrebbero appartenervi
    - es. `House` insieme di tutte le cose, anche future, classificabili come *case*
- **Proprietà** — *relazioni* che descrivno individui associandoli a:
  - *dati*, *valori* di tipi predefiniti, come gli interi o le stringhe → **datatype property**
    - es. `streetName` può associare vie a stringhe (loro nomi)
  - altri *individui* → **object property**
    - es. `nextTo` relazione tra case e `onStreet` tra case e vie

OWL ha alcune varianti che differiscono nelle restrizioni che è possibile imporre su classi e proprietà e nell'implementazione:

- in **OWL-DL** una classe non può essere un individuo o una proprietà e una proprietà non è un individuo
- in **OWL-Full**, individui, proprietà e classi *non* necessariamente *disgiunti*
- OWL2 ha 3 *profili* orientati verso specifiche applicazioni che limitano i costrutti usabili, per garantire l'efficienza dell'*inferenza*
  - **OWL2 EL** consente descrizioni con molti dettagli strutturali
    - es. utili alle grandi ontologie biomediche
  - **OWL2 QL** pensato come front-end per i *linguaggi di interrogazione* per DB
  - **OWL2 RL** progettato per l'uso di regole

Default

- OWL non assume la *UNA*:
  - due nomi non denotano necessariamente individui o classi diversi
- OWL non assume *conoscenza completa*:
  - non si può assumere che tutti i fatti rilevanti siano stati dichiarati

Principali **classi predefinite** e **costruttori di classe** di OWL:

- C_i classi, P proprietà, ***tripla*** xPy = atomo $P(x, y)$

Classe	Contiene
`owl:Thing`	tutti gli individui
`owl:Nothing`	nessun individuo
`owl:ObjectIntersectionOf`$(C_1, \dots, C_k)$	individui in $C_1 \cap \cdots \cap C_k$
`owl:ObjectUnionOf`$(C_1, \dots, C_k)$	individui in $C_1 \cup \cdots \cup C_k$
`owl:ObjectComplementOf`(C)	individui non in C
`owl:ObjectOneOf`$(I_1, \dots, I_k)$	$I_1, \dots, I_k$
`owl:ObjectHasValue`(P, I)	$\{x \mid xPI\}$
`owl:ObjectAllValuesFrom`(P, C)	$\{x \mid xPy \rightarrow y \in C\}$
`owl:ObjectSomeValuesFrom`(P, C)	$\{x \mid \exists y \in C : xPy\}$
`owl:ObjectMinCardinality`(n, P, C)	$\{x \mid \#\{y \mid xPy \wedge y \in C\} \geq n\}$
`owl:ObjectMaxCardinality`(n, P, C)	$\{x \mid \#\{y \mid xPy \wedge y \in C\} \leq n\}$
`owl:ObjectHasSelf`(P)	$\{x \mid xPx\}$

OWL ha **predicati predefiniti** con interpretazione fissata (alcuni presenti anche in RDF, RDF-S):

notazione funzionale per OWL: *significato*

nel seguito, x e y universalmente quantificate

- `rdf:type`(I, C), `owl:ClassAssertion`(C, I): $I \in C$
- `rdfs:subClassOf`(C_1, C_2), `owl:SubClassOf`(C_1, C_2): $C_1 \subseteq C_2$
- `rdfs:domain`(P, C), `owl:ObjectPropertyDomain`(P, C): se xPy allora $x \in C$
- `rdfs:range`(P, C), `owl:ObjectPropertyRange`(P, C): se xPy allora $y \in C$
- `owl:EquivalentClass`$(C_1, C_2, \ldots, C_k)$: $C_i \equiv C_j$ per ogni i, j
- `owl:DisjointClass`$(C_1, C_2, \ldots, C_k)$: $C_i \cap C_j = \emptyset$ per ogni $i \neq j$

- `rdfs:subPropertyOf`(P_1, P_2): xP_1y implica xP_2y
- `owl:EquivalentObjectProperties`(P_1, P_2): xP_1y sse xP_2y
- `owl:DisjointObjectProperties`(P_1, P_2): xP_1y implica che non vale xP_2y
- `owl:InverseObjectProperties`(P_1, P_2): xP_1y sse yP_2x
- `owl:SameIndividual`$(I_1, \ldots, I_n)$: $\forall j \forall k\ I_j = I_k$
- `owl:DifferentIndividuals`$(I_1, \ldots, I_n)$: $\forall j \forall k\ j \neq k$ implica $I_j \neq I_k$
- `owl:InverseFunctionalObjectProperty`(P): se x_1Py e x_2Py allora $x_1 = x_2$
- `owl:TransitiveObjectProperty`(P): se xPy e yPz allora xPz
- `owl:SymmetricObjectProperty`(P): se xPy allora yPx
- `owl:AsymmetricObjectProperty`(P): xPy implica che non vale yPx
- `owl:ReflectiveObjectProperty`(P): xPx per ogni x
- `owl:IrreflectiveObjectProperty`(P): non vale xPx per ogni x

- altre sintassi: XML, TURTLE, ...

Esempio — Costruttori di classe (sintassi funzionale):

- `ObjectHasValue(lc:has_logo lc:lemon_icon)`
  - classe di oggetti per i quali la proprietà `lc:has_logo` ha come valore l'individuo `lc:lemon_icon`
- `ObjectSomeValuesFrom(lc:has_color lc:green)`
  - classe di oggetti che hanno per colore una tonalità di verde
    - `lc:green` classe composta da tonalità specifiche, come smeraldo od olivastro
- `MinCardinality(2 :owns :building)`
  - classe di tutti gli individui che possiedono due o più edifici:

$$\{x \mid \exists i_1 \exists i_2 \; building(i_1) \wedge building(i_2) \wedge owns(x, i_1) \wedge owns(x, i_2) \wedge i_1 \neq i_2\}$$

I costruttori di classe vanno usati in *enunciati*:

- ad es. per dire che un individuo è membro di tale classe
  o che una classe sia equivalente a un'altra

OWL NON esprimibile attraverso clausole definite:

- per dire che tutti gli elementi di S hanno il valore v per un dato predicato p,
  si dice che S è sottoinsieme di quello di tutte le cose con valore v per p
  (*assioma di inclusione*, $\sqsubseteq$)

Esempio — Di seguito la rappresentazione in sintassi OWL funzionale della rete semantica vista in precedenza

```
Prefix(lc:=<http://artint.info/ontologies/lemon_computers.owl#>)
Ontology(<http://artint.info/ontologies/lemon_computers.owl>

 Declaration(Class(lc:computer))
 Declaration(Class(lc:logo))
 ClassAssertion(lc:logo lc:lemon_icon)
 Declaration(ObjectProperty(lc:has_logo))
 ObjectPropertyDomain(lc:has_logo lc:computer)
 ObjectPropertyRange(lc:has_logo lc:logo)

 Declaration(Class(lc:lemon_computer))
 SubClassOf(lc:lemon_computer lc:computer)
 SubClassOf(lc:lemon_computer
            ObjectHasValue(lc:has_logo lc:lemon_icon))
```

```
Declaration(Class(lc:color))
Declaration(Class(lc:green))
Declaration(Class(lc:yellow))
SubClassOf(lc:green lc:color)
SubClassOf(lc:yellow lc:color)
Declaration(Class(lc:material_entity))
SubClassOf(lc:computer lc:material_entity)
ObjectPropertyDomain(lc:has_color lc:material_entity)
ObjectPropertyRange(lc:has_color lc:color)
SubClassOf(lc:lemon_computer
           ObjectSomeValuesFrom(lc:has_color lc:green))
SubClassOf(lc:lemon_computer
           ObjectSomeValuesFrom(lc:has_color lc:yellow))

)
```

- prima riga: `lc:` abbreviazione → `lc:computer` per l'URI `<http://artint.info/ontologie/lemon_computers.owl#computer>`
- seconda riga: introduce l'ontologia
- `lc:computer` e `lc:logo` classi
- `lc:lemon_icon` membro della classe `lc:logo`
- `lc:has_logo` proprietà con dominio `lc:computer` e codominio `lc:logo`
- per dichiarare che tutti i computer Lemon hanno per logo l'icona di un limone: insieme dei computer Lemon incluso nell'insieme di tutte le cose per le quali la prop. `has_logo` ha valore `lemon_icon`
- `lc:lemon_computer` sottoclasse di `lc:computer` e dell'insieme di individui con valore `lc:lemon_icon` per `lc:has_logo`
  - cioè, tutti i computer Lemon hanno l'icona d'un limone come logo
- verde e giallo sottoclassi di `color`
  - `has_color` si applica a entità materiali, ossia oggetti fisici
  - alcuni dei colori di un computer Lemon sono giallo e verde

Altri Costrutti

Costruttore di proprietà in OWL:

- `owl:ObjectInverseOf(P)`
  - per la proprietà inversa di P, denotata anche con P^{-1}: $yP^{-1}x$ sse xPy
  - solo per *object-property*:
    - le *datatype-property* non hanno proprietà inverse:
      valori di *tipi concreti* non possono fungere da soggetto di triple

---

Classi *datatype* corrispondenti ai codomini di proprietà

- `owl:DataSomeValuesFrom` e `owl:EquivalentDataProperties` con def. analoghe a quelle per proprietà su individui

Costrutti per definire proprietà, *commenti*, *annotazioni*, *versioning* e *import* da altre ontologie

Esempio — Si consideri la definizione di *apartment building*:

- edificio residenziale con più unità abitative che possono essere affittate
  - in contrasto con: *condominio*, dove le unità vengono vendute individualmente; *casa*, in cui c'è solo un'unità
- data `ResidentialBuilding` sottoclasse di `Building`:
  - definizione funzionale della proprietà `numberOfUnits` con dominio `ResidentialBuilding` e codominio `{one,two,moreThanTwo}`
    - analogamente per la proprietà `ownership` con dominio `ResidentialBuilding` e codominio `{rental,ownerOccupied,coop}`

```
Declaration(ObjectProperty(:numberOfunits))
FunctionalObjectProperty(:numberOfunits)
ObjectPropertyDomain(:numberOfunits :ResidentialBuilding)
ObjectPropertyRange(:numberOfunits
                    ObjectOneOf(:two :one :moreThanTwo))
```

(cont.)

- quindi un `ApartmentBuilding` è un `ResidentialBuilding` in cui `numberOfUnits` ha valore `moreThanTwo` e `ownership` ha valore `rental`
  - definite le classi di individui che hanno un valore per ciascuna proprietà, `ApartmentBuilding` equivarrà alla loro intersezione:

```
Declaration(Class(:ApartmentBuilding))
EquivalentClasses(:ApartmentBuilding
    ObjectIntersectionOf(
        :ResidentialBuilding
        ObjectHasValue(:numberOfunits :moreThanTwo)
        ObjectHasValue(:ownership :rental)))
```

- usabile per rispondere a domande circa gli `ApartmentBuilding`, loro proprietà e sul numero di unità
- `ApartmentBuilding` ***eredita*** le proprietà di `ResidentialBuilding`

Ontologie di Dominio

Una **ontologia di dominio** riguarda un particolare dominio d'interesse:

- molte delle ontologie esistenti riguardano un dominio ristretto definito per specifiche applicazioni

Linee Guida per definire ontologie di dominio

- Se possibile, usare ontologie *esistenti*
  - la KB potrà interagire con altre che le adottano
- Se esiste un'ontologia che non corrisponda esattamente ai requisiti, la si può *importare* facendo poi delle *aggiunte*
  - non si parte da zero
  - se la propria ontologia include e migliora l'altra, altri la vorranno adottare, aumentando l'interoperabilità della loro applicazione
- Assicurarsi che l'ontologia *si integri* con ontologie affini
  - ad es. ontologia su resort da integrare con altre su cibi, spiagge, sport, ecc.
  - tentare di assicurare l'uso della stessa terminologia per le stesse cose

- Tentare di conformarsi a ontologie di *livello superiore*
  - rende molto più facile l'integrazione della conoscenza di/con altre fonti
- Nel progettare una nuova ontologia, consultarsi con altri *utenti potenziali*
  - la rende più utile e più facilmente adottabile
- Seguire le *convenzioni* di denominazione
  - ad es. chiamare una classe con il nome singolare dei sui membri: `Resort` e non `Resorts` o peggio `ResortConcept`
  - pensare all'uso di classi e proprietà
    - meglio dire che "`r1` è di tipo `Resort`" che "`r1` è di tipo `Resorts`", ancor meglio di "`r1` è di tipo `ResortConcept`"
- Specificare la corrispondenza tra ontologie:
  - a volte serve un allineamento tramite *matching* di ontologie sviluppate in modo indipendente
  - da evitare, se rende la conoscenza molto più complicata / ridondante

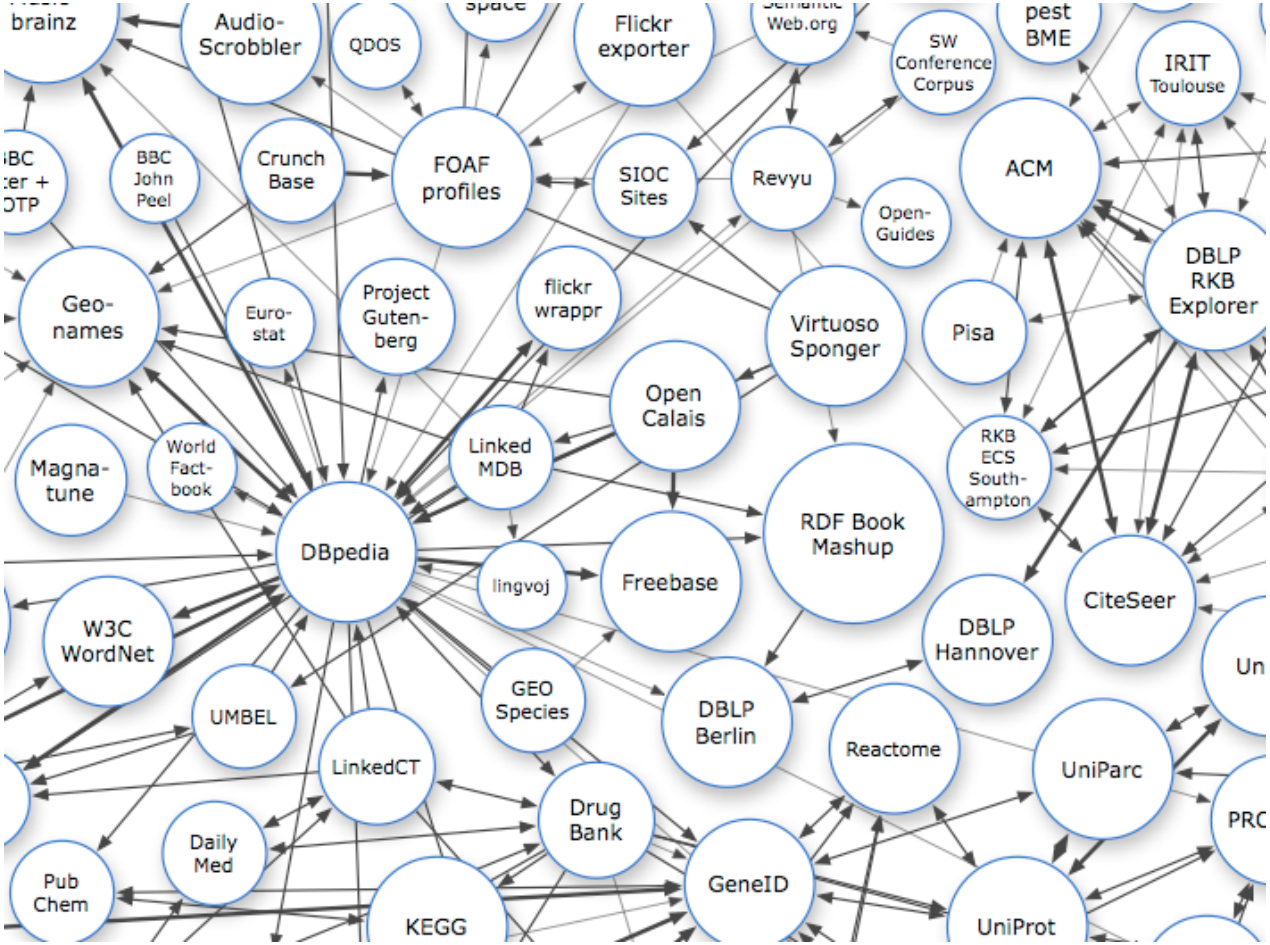
Editor OWL

- Editor di ontologie, come Protégé:
  - fornisce un modo per definire ontologie a un *livello di astrazione* appropriato
  - volendo usare un concetto, ne facilita la *ricerca* nella terminologia o può segnalarne l'eventuale assenza
  - aiuta a determinare immediatamente il *significato* di un termine
  - permette controlli di *correttezza* sulle ontologie
    - corrispondenza con l'interpretazione intesa per i termini
  - facilità la *riusabilità*
    - dovrebbe usare il più possibile un linguaggio standard

Web of Data

Prospettiva del Web Semantico — sfrutta l'infrastruttura del Web per creare grafo globale di sorgenti *distribuite* di dati collegati

- loro semantica disponibile anche alle macchine sulla base di ontologie RDF/OWL
  - oltre alle query SPARQL, veri servizi di *ragionamento* automatico (tramite *reasoner*)
- server come *triple store* o altri DB NoSQL
  - e.g. document DB basati su JSON-LD



Da http://linkeddata.org

Princìpi dei **Linked Data** [10]

1. *URI* come nomi per le cose
2. *URI HTTP* per permetterne la ricerca
    - sull'infrastruttura del Web
3. informazione fornita usando *standard*
    - RDF, SPARQL, ...
4. *link* verso altri URI
    - abilitano la scoperta di altra conoscenza

Qualità degli *Open Data*: categorizzazione 1-5 stelle di TBL [10]

5✪ — Linked Open Data

- navigabili attraverso strumenti come LodLive
    - es. LOD su film girati in Puglia

Riferimenti

[1] D. Poole, A. Mackworth: *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press [Ch.14]
[2] D. Poole, A. Mackworth, R. Goebel: *Computational Intelligence: A Logical Approach*. Oxford University Press
[3] S. J. Russell, P. Norvig: *Artificial Intelligence* Pearson. 4th Ed. (ch. 9-10) - cfr. anche ed. Italiana
[4] J. Sowa: *Knowledge Representation: Logical, Philosophical, and Computational Foundations* Brooks Cole/Cengage
[5] R. J. Brachman and H. J. Levesque: *Knowledge representation and reasoning*. Morgan Kaufmann. (2004)
[6] T. Berners-Lee, J. Hendler and O. Lassila: *The semantic web: a new form of web content that is meaningful to computers will unleash a revolution of new possibilities*. Scientific American May, pp. 28–37 (2001)
[7] K. Janowicz, F. van Harmelen, J. A. Hendler and P. Hitzler: *Why the data train needs semantic rails*. AI Magazine 36 (1), pp. 5–14. (2015)
[8] R. A. Kowalski: *Logic for problem solving, revisited*. Books on Demand. (2014)
[9] P. Hitzler, M. Krötzsch, S. Rudolph: *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC (2009) [sito]
[10] T. Heath & C. Bizer *Linked Data: Evolving the Web into a Global Data Space* Morgan & Claypool (2011) [sito]
[11] A. Hogan, et al.: *Knowledge Graphs*. ACM Comput. Surv. 54(4): 71:1-71:37 (2021)

LINK

Semantic Web presso il W3C

- IRI
- RDF e RDF-Schema
  - Notation3 (N3)
  - N-Triples
  - Turtle
  - TriG
- OWL2
- SWRL *linguaggio a regole* basato su OWL e RuleML

- Interrogazione
  - SPARQL
- Inferenza
- Linked Data
  - linkeddata.org
  - Linked Open Vocabularies
  - wikidata
  - DBpedia
    - endpoint

[DL] Description Logics Homepage
[Protégé] Protégé standalone o anche via Web

[JSON-LD] sito-base
[Owlready] Owlready2 — libreria Python per OWL/RDF

NOTE

[⋖] consigliata la lettura
[versione] 7/11/2022, 09:38:54

Figure tratte da [1] salvo diversa indicazione