Università degli Studi di Bologna

Dottorato di Ricerca in Ingegneria Elettronica e Informatica
VII ciclo (A.A. '91/'92, '92/'93, '93/'94)

Programmazione Logica Orientata agli Oggetti

Tesi di Dottorato di

dott. ing. Andrea Omicini

Relatore

prof. ing. Antonio Natali

Coordinatore

prof. ing. Bruno Riccò

Febbraio 1995

Indice Generale

0 Introduzione

0.1 Considerazioni preliminari

0.1.1 La produzione in larga scala del software

Le tecnologie della produzione industriale del software non hanno ancora dimostrato la loro capacità di mettere i programmatori al passo con l'evoluzione tecnologica dell'hardware. Mentre gli elaboratori sono sempre più veloci, e le relative tecniche di sviluppo consentono ormai di passare dal progetto alla produzione nel giro di pochi mesi, il campo del software ha a che fare con tempi di sviluppo sempre più onerosi, e con prestazioni che sovente non rendono giustizia alle capacità di elaborazione delle macchine.

Per questo, molte energie vengono attualmente spese nel campo

dell'ingegneria del software, cercando di individuare gli approcci e le metodologie che possano contribuire a ridurre gli effetti deleteri e crescenti del "collo di bottiglia" rappresentato dalla produzione di programmi di grandi dimensioni.

Le tipologie di intervento possono essere classificate grossolanamente in due classi. La prima tende a elevare quanto più possibile il grado d'intelligenza della macchina virtuale, riducendo quindi i dettagli di cui il programmatore si deve occupare. È questa la strada battuta dai sistemi di prototipazione, dai linguaggi di specifica eseguibili, e dai linguaggi logici in particolare.

La seconda, invece, tende a disciplinare il lavoro del programmatore, in modo da guidare le fasi di progetto e sviluppo di un sistema software secondo metodologie standardizzate e di massima produttività, in termini di leggibilità dei programmi, modificabilità, riusabilità dei componenti, ecc.. È questa la strada battuta principalmente dai linguaggi orientati agli oggetti, e da tutte le metodologie di sviluppo correlate al modello a oggetti.

Risulta quindi naturale volgere la propria attenzione ai paradigmi logico e orientato agli oggetti quali candidati ideali per nuovi modelli e linguaggi di programmazione che possano divenire potenti strumenti per lo sviluppo del software su larga scala.

0.1.2 I paradigmi di programmazione logica e a oggetti

La programmazione logica e quella orientata agli oggetti vengono sovente poste in contrasto come paradigmi tra loro incompatibili. In particolare, si fa rilevare come i linguaggi logici non posseggano meccanismi flessibili per la strutturazione dei programmi, cosa che si riflette in una capacità di modellazione dei domini applicativi non adeguata, quando la si confronti con quella propria dei linguaggi a oggetti [Weg92b]. Per colmare questo gap, sono state avanzate diverse proposte ([McC92, AnPa90, Zan84], tra le tante), che si concentrano usualmente sulla definizione di oggetto nella

programmazione logica, e integrano alcune delle caratteristiche base dei due paradigmi. Tuttavia, tali proposte non riescono in genere a soddisfare tutti i requisiti richiesti affinché il modello risultante possa essere definito contemporaneamente e a pieno titolo sia logico sia orientato agli oggetti.

È d'altra parte unanimemente riconosciuto l'impatto che entrambe le classi di linguaggi hanno sullo sviluppo del software. Adottando un approccio alla modellazione dei domini applicativi che è orientato ai dati (*data-oriented*) piuttosto che alle procedure (*procedure-oriented*), i linguaggi a oggetti promuovono nuove tecniche di progetto e sviluppo del software, che consentono la specifica incrementale, la prototipazione, la condivisione e il riuso del codice.

D'altra parte, un linguaggio logico è dotato dell'intrinseca capacità di ridurre la distanza tra la fase di specifica e quella di codifica di un programma, data la sostanziale equivalenza della sua semantica dichiarativa ed operazionale, che ne fa una sorta di linguaggio di specifica eseguibile.

Da ciò risulta chiaro che costruire un archetipo di linguaggio di programmazione dichiarativo con modello dei dati a oggetti può portare a risultati notevoli nel campo dell'ingegneria del software.

0.1.3 Scopo della tesi

Lo scopo di questa tesi è quindi quello di definire un unico modello computazionale basato sulla logica del prim'ordine, che sia in grado di catturare tutto quanto è essenziale nei paradigmi logico e orientato agli oggetti, senza perdere nessuna delle caratteristiche fondamentali di ciascuno dei due approcci.

Dopo un'analisi delle soluzioni disponibili, la scelta di una rappresentazione degli oggetti come teorie logiche emergerà come la più utile; in particolare si potrà mostrare come la più parte delle caratteristiche peculiari del paradigma a orientato agli oggetti possa essere catturata e riprodotta in ambito logico adottando ed estendendo opportunamente il modello della programmazione

logica contestuale. Il modello risultante verrà poi descritto, e ne sarà discussa una realizzazione in termini sia di linguaggio sia di implementazione.

I limiti intrinseci del modello introdotto (relativi in particolare alla capacità di esprimere tanto la nozione di stato di un oggetto, quanto quella di relazione classe/istanza) fungeranno poi da linee-guida per un nuovo framework concettuale basato (*i*) sull'abduzione, per la creazione e configurazione dinamica di teorie logiche in modo dichiarativo, e (*ii*) sul concetto di vincolo di metalivello per la definizione di un modello computazionale adeguato.

L'introduzione del concetto di computazione a vincoli consentirà di porre in evidenza come programmare per relazioni con un modello dei dati a oggetti può in linea di principo consentire la riconciliazione di metodologie di programmazione apparentemente distanti come quella a oggetti e quella a vincoli [JaLa87, JaMa94], in un quadro concettuale più semplice e uniforme (e quindi, in breve, più convincente) di altri approcci allo stesso problema noti in letteratura [Bor87, FrBe90].

0.2 Struttura della tesi

Il capitolo 1 è dedicato a illustrare in breve le principali caratteristiche dei due paradigmi, logico e orientato agli oggetti, in modo da definire meglio le basi concettuali su cui poggia la dissertazione.

Il capitolo 2 discute brevemente le diverse alternative disponibili per un modello dei dati a oggetti in programmazione logica. Con l'ausilio di semplici esempi, vengono evidenziati i punti di forza e le debolezze propri dei diversi approcci.

Il capitolo 3 tratta delle caratteristiche dell'ambiente di programmazione logica contestuale CSM [CSM]. CSM è una implementazione funzionante di un'estensione [DNO94] del paradigma di programmazione logico contestuale [MoPo89, MeNa92], costruita sulla base del framework SICStus Prolog [SICS]. Tale realizzazione ha rappresentato il primo risultato (sia concettuale

sia implementativo) nella direzione di un ambiente di programmazione logico orientato gli oggetti. In particolare, il capitolo ne discute il modello concettuale e computazionale, presentandone la connotazione semantica sia in senso dichiarativo sia in senso operazionale. Infine, ne vengono discussi i limiti intrinseci.

Il capitolo 4 introduce l'idea di configurazione di stato di un oggetto come trattamento di conoscenza incompleta, sviluppando il ruolo del ragionamento abduttivo come strumento concettuale per la creazione e la configurazione dichiarativa di teorie logiche. Si discute come estendere il framework abduttivo classico verso un modello multiteoria e con estensione di programma.

Il capitolo 5 discute il concetto di vincolo di metalivello, descrivendo un percorso verso la definizione di un paradigma di programmazione logico che possa dirsi a pieno titolo orientato agli oggetti. Viene discusso in linea di principio un approccio alla definizione del modello computazionale, strettamente imparentato con gli schemi della programmazione a vincoli, e basato sul concetto di (meta)vincolo di dimostrabilità. Un linguaggio campione viene introdotto a titolo d'esempio, e sono discusse alcune computazioni particolarmente significative.

Il capitolo 6 è dedicato alle conclusioni e ai possibili sviluppi dei temi di ricerca proposti in questa tesi.

L'appendice A accenna brevemente alcuni dei fondamenti teorici della logica del prim'ordine e dei suoi linguaggi.

L'appendice B, infine, estende le basi della logica del prim'ordine a coprire i concetti di formula e teoria logica etichettata, necessari quali fondamenti per le soluzioni proposte nel capitolo 4.

0.3 Ringraziamenti

Un lavoro di tre anni (e più) può difficilmente essere contenuto in un unico

esauriente trattato. Questa tesi ha comunque l'obiettivo di raccogliere intorno a un nucleo centrale coerente una parte consistente del variegato lavoro di ricerca da me svolto in questi anni al Dipartimento di Elettronica, Informatica e Sistemistica.

Un'attività di tale durata, intensità e soddisfazione non sarebbe stato possibile senza l'aiuto di tante persone che lavorano o hanno lavorato al DEIS, e che non possono tutte essere ricordate per ovvie ragioni in questa sede.

Primo tra tutti, vorrei ricordare l'incessante sostegno del relatore di questa tesi, Antonio Natali, la cui energia, competenza e fiducia mi hanno accompagnato in ogni passo della mia ricerca. Le infinite discussioni intorno al suo tavolo sono state la fonte primaria di tutte le idee raccolte in questo lavoro.

Collaborazione feconda, e amicizia sincera, mi è sempre giunta da Enrico "Lugano" Denti, paziente e diligente compagno in molte fasi della mia attività. Irrinunciabile il contributo dato poi dai tanti tesisti con cui ho avuto la fortuna di lavorare: da Paolo "Laser" Lasagni a Pier Alberto Guidotti, da Davide Bolcioni a Marco "Daemon" Venuti.

Evelina Lamma e Paola Mello sono state (insieme ad Antonio Natali) gentili e disponibili "giganti" sopra le cui spalle ho potuto salire. Tanti hanno poi offerto il destro a importanti approfondimenti, con obiezioni e idee: tra gli altri, Michele Bugliesi, Antonio Porto, e Luis Monteiro. Molti, infine, coloro che con la loro amicizia, pazienza e disinteressato supporto hanno reso più felice la mia vita al DEIS: tra tutti, Cesare Stefanelli, Franco Zambonelli, Michela Milano, Letizia Leonardi, Antonio Corradi, Anna Ciampolini, Maurelio Boari, Rita Mambelli e gentili colleghe, Zsolt Kovacs, Luca Simoncini, Alberto Leone.

Il mio lavoro triennale ha poi avuto il triennale supporto del Consiglio Nazionale delle Ricerche nell'ambito del Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo".

Tutta la mia famiglia, infine, ha con il suo conforto reso la mia vita sempre serena: i miei genitori Albertina e Ilario, mia sorella Grazia, e poi Siria e Luciano, Luciana e Nello, Barbara e Daniele, Mirko, Annalisa e Massimiliano, Irene e Franco, e tutti gli altri. In particolare, vorrei dedicare questo lavoro a tre generazioni diverse della mia famiglia. Ai miei nonni, in primo luogo: Aurelio, Domenica, Ornella e Nino, che hanno sempre dato alla cultura e al sapere il peso dovuto anche nelle situazioni più avverse. Ai miei nipoti, in secondo luogo: Manuel e Paco, Emanuela e Leonardo, il cui futuro è la ragione vera per cui vale la pena di fare ricerca.

Alla dolcissima Sara, mia moglie, infine: la seconda più grande fortuna della mia vita è averla incontrata. La prima, accadimento le cui ragioni restano per molti avvolte nel più fitto mistero, è che abbia voluto essermi accanto per la vita.

1 Programmazione logica e orientata agli oggetti

1.1 Programmazione logica

1.1.1 Cos'è la programmazione logica?

Non esiste un accordo generale in letteratura sul tema di che cosa definisca compiutamente il concetto di linguaggio logico. Ogni autore tende a dare la definizione che più si confà al prodotto della sua ricerca. Non potendo quindi per mancanza di riferimenti certi sfuggire a tale infausta regola, cercheremo se

non altro di chiarire preventivamente l'ottica che ci porta a utilizzare un criterio piuttosto che un altro.

Poiché la motivazione primaria che ci spinge a cercare un modello unitario per la programmazione logica a oggetti è la ricerca di un paradigma che consenta di ridurre al massimo i tempi per la progettazione e lo sviluppo di sistemi software di grandi dimensioni, l'ottica adottata è quella tipica dell'ingegneria del software.

Da questo punto di vista, un primo ovvio requisito per un linguaggio che voglia dirsi logico risulta l'esistenza di una semantica dichiarativa equivalente a quella operazionale: ciò che ne consentirebbe la qualifica di linguaggio di specifica eseguibile. Inoltre, ovvie considerazioni di carattere pratico (mai sufficientemente prese in considerazione a livello accademico) consigliano di seguire una strada già battuta da strumenti diffusi a livello industriale: basarsi sulla logica del prim'ordine, e perseguire la massima compatibilità con l'unico linguaggio logico industrialmente disponibile in maniera diffusa, cioè il Prolog, pare in questo senso un percorso quasi obbligato.

Da quest'ultimo punto di vista, il fondamento sintattico di un linguaggio logico non è una scelta indifferente rispetto agli scopi proposti: il formalismo tipico del Prolog è quindi una sorta di riferimento obbligato. Un programma scritto in un linguaggio logico come Prolog è una collezione di *clausole di Horn*, ossia un insieme di particolari formule della *logica del prim'ordine*.[1]

Una clausola di Horn in un programma logico si trova di solito nella sua forma classica:

$$H \leftarrow B_1, B_2, \ldots, B_n$$

ove H è la testa della clausola, e $B_1, B_2, \ldots, B_n$ rappresenta il *corpo* della

---

1 Per una descrizione formale e completa della logica del prim'ordine, si veda [CLM91]. Per un quadro generale dei concetti e dei risultati fondamentali riguardanti logica matematica e teorie formalizzate, si consulti per esempio [Rog78]. Alcuni cenni sui fondamenti utilizzati in questa sede sii possono trovare anche nell'appendice A.

clausola stessa. Una *clausola unitaria*, o *fatto*, è una clausola di Horn con il corpo vuoto (n=0), la cui forma è quindi

$H \leftarrow$

La prima lettura che di un programma logico si può fare è quella di una collezione di assiomi logici: ciò corrisponde all'*interpertazione dichiarativa* di un programma. Infatti, tramite quelle particolari formule logiche che sono le clausole, il programmatore dichiara le proprietà del dominio applicativo, nei termini delle relazioni sussistenti tra gli oggetti del dominio medesimo.

Peraltro, la particolare forma delle clausole di Horn si presta a illustrare la proprietà che consente alla logica del prim'ordine di essere usata come vero e proprio linguaggio di programmazione: la sua *intepretazione procedurale* [Ko74]. Infatti, una clausola di Horn può essere vista come la definizione di una procedura scritta in un linguaggio di programmazione logico. Secondo tale interpretazione, la testa della clausola rappresenta l'intestazione della procedura, mentre il corpo della clausola costituisce la definizione della procedura stessa, in termini di chiamate ad altre procedure. Il passaggio dei parametri avviene sfruttando il meccanismo dell'*unificazione*, che diviene meccanismo uniforme anche per la selezione e per la costruzione dei dati.

In programmazione logica, quindi, il concetto di deduzione logica sussume quello di computazione. Data una congiunzione di atomi logici (il *goal iniziale*), un sistema logico è in generale in grado di calcolarne effettivamente il valore di verità (ossia, di *dimostrare* il goal) rispetto a una data *teoria logica*, ossia la collezione degli assiomi che costituiscono il programma. Tale valore di verità è in generale espresso in forma condizionata rispetto all'istanziazione di variabili del goal iniziale, secondo la cosiddetta *risposta calcolata*. Ad esempio, l'atomo logico `p(X)` è vero rispetto alla teoria logica `{p(X)←q(X), q(f(a))}` se `X=f(a)`.

Da questo punto di vista, il procedimento inferenziale che conduce alla risposta calcolata può essere visto come una forma di *dimostrazione*

costruttiva: la prova del goal iniziale conduce alla configurazione di oggetti che costituiscono il risultato della computazione. Ad esempio, la prova di `p(X)` rispetto alla teoria logica {`p(X)`←`q(X)`, `q(f(a))`} porta alla configurazione di `X` come l'oggetto `f(a)`.

L'idea fondamentale della programazione logica, secondo [Kow74], è la separazione ideale tra le due componenti di un algoritmo: la *logica* della soluzione del problema, ed il *controllo*, ossia la sequenza delle operazioni che rendono effettivo un algoritmo. Il programmatore logico, in teoria, dovrebbe abbandonare la visione *procedurale* degli algoritmi (ciò che il programma deve effettivamente fare) per concentrarsi sulla *descrizione* delle caratteristiche del problema, degli elementi che lo compongono e delle relazioni che tra questi intercorrono. È il sistema di programmazione logica che, idealmente, deve farsi carico della parte operazionale, ovvero del controllo.

L'approccio dichiarativo, infatti, si fonda sulla definizione di ciò che risulta vero nel dominio del discorso: sia questo rappresentato da verità immutabili proprie del dominio (es., $A < A+1$ per ogni A intero), o da proprietà tipiche del particolare problema da risolvere (es., $X \leq Y+3$ per qualche particolare X e Y dati). Sta poi al sistema (e non al programmatore) determinare la sequenza di operazioni che costituiscono la procedura di soluzione del problema specifico (es., determinare la soluzione di un sistema di disequazioni intere).

Corrispondentemente alla separazione concettuale tra logica e controllo, e alla doppia interpretazione come linguaggio di specifica e di programmazione, la caratterizzazione semantica di un linguaggio logico viene usualmente condotta lungo due principale direttive: la *semantica operazionale*, che definisce la procedura di dimostrazione di un goal rispetto a un programma scritto nel linguaggio, e la *semantica dichiarativa*, che denota un programma (ossia, una teoria logica) nei termini dell'insieme delle verità che possono essere dimostrate a partire da esso.

Intuibilmente, la più importante caratteristica di un linguaggio logico non

è tanto la doppia caratterizzazione semantica che gli si può attribuire, quanto, invece, il fatto che queste due caratterizzazioni devono ovviamente essere equivalenti: quanto risulta dimostrabile a partire da un programma applicando la procedura di prova deve nel contempo essere parte dell'insieme delle verità che denotano dichiarativamente il programma stesso.

Mentre l'aspetto dichiarativo rimanda all'idea di un linguaggio logico come linguaggio di specifica, la caratterizzazione procedurale garantisce l'eseguibilità dei suoi programmi: la coesistenza delle due caratterizzazioni semantiche ne garantisce quindi l'interpretazione come linguaggio di specifica eseguibile.

Infine, altre questioni fondamentali quali l'*interpretazione a processi*, o l'*interpretazione come basi di dati*, intrinseche ai linguaggi logici, non sono rilevanti dal punto di vista di questa tesi: la scelta del Prolog come linguaggio di riferimento porta a considerare quale punto di partenza un modello di linguaggio logico sequenziale, che utilizzi la semplice risoluzione lineare con regola di calcolo *left-most* (che seleziona l'atomo più a sinistra). Ciò comporta la possibilità concettuale di implementare il modello risultante su un supporto diffuso come SICStus Prolog senza stravolgerne l'impianto, e sfruttandone quindi al massimo l'efficienza e gli strumenti che esso mette a disposizione.

1.1.2 Vantaggi della programmazione logica

Il principale campo di applicazione della programmazione logica è costituito in generale da tutti gli ambiti applicativi in cui è richiesta una elevata capacità di elaborazione simbolica. La capacità di fare inferenze rende i sistemi basati sulla logica i più adatti a realizzare sistemi dotati di capacità di ragionamento.

Quello dell'intelligenza artificiale, per quanto storicamente campo del paradigma funzionale, è pertanto il campo che probabilmente meglio si attaglia alle capacità di elaborazione proprie dei linguaggi logici.

Inoltre, l'uniformità tra rappresentazione del codice e dei dati, tipica dei linguaggi logici come Prolog, li rende particolarmente adatti all'utilizzo di

tecniche di *metaprogrammazione*.

Dal punto di vista della manutenibilità e modificabilità del codice, i linguaggi logici offrono indubbiamente molto: l'esistenza di un meccanismo uniforme per il passaggio dei parametri, la selezione e la costruzione dei dati (l'unificazione), l'uniformità di codice e dati, la trasparenza referenziale, e l'assenza di effetti collaterali, rendono i singoli frammenti di un programma logico generalmente molto più leggibili del codice scritto in altri tipi linguaggi.

In particolare, il meccanismo di unificazione consente di trattare *oggetti non completamente specificati*: non è necessario per esempio assegnare un valore a una variabile per poterla usare in una computazione. Ciò elimina la necessità di quella fase di *inizializzazione* dei dati tipica dei linguaggi imperativi.

Infine, le caratteristiche di linguaggi di specifica eseguibili proprie dei linguaggi logici promuovono un approccio alla costruzione del software tipicamente *top-down*. Essi consentono infatti di scrivere prototipi di programma che possono essere eseguiti, e quindi successivamente sviluppati in un processo di raffinamento incrementale. Ciò rende i linguaggi logici strumenti ideali per lo sviluppo incrementale e veloce del software.

1.1.3 Limiti della programmazione logica

Il principale problema dei linguaggi logici come strumenti per lo sviluppo del software risiede nell'idea di programma monolitico, di unica teoria logica che descrive compiutamente tutte le proprietà del dominio applicativo. La teoria che costituisce un programma logico è costituita da verità *universali* ed *immutabili*: la sua monoliticità è quindi sia *spaziale* sia *temporale*, rispettivamente.

L'incapacità di modellare opportunamente in termini di strutture di programma la multiformità del dominio ha portato molti Prolog commerciali a introdurre un concetto di modulo come contenitore sintattico per costruire programmi logici a teorie multiple. La mancanza di caratterizzazione semantica

dei moduli rende però insoddisfacente tale soluzione.

D'altra parte, l'immutabilità degli assiomi di un programma logico lo rende virtualmente incapace di modellare opportunamente i domini dipendenti dal tempo. In particolare, risulta difficile associare a un oggetto del dominio applicativo un'appropriata nozione di stato.

L'intrinseca mancanza di un sistema di tipi dei linguaggi logici, poi, se da una parte è una caratteristica positiva, dall'altra impedisce un qualunque tipo di controllo dei programmi basato sui tipi, rendendo così più ostico il rilevamento e la correzione di eventuali errori di programmazione.

Infine, la regola di ricerca propria del Prolog risulta sovente limitante, in particolare nella risoluzione di problemi complessi, quali quelli tipici della ricerca operativa: da ciò emerge la necessità di dotare i sistemi logici della capacità di realizzare strategie di ricerca più evolute.

1.1.4 Programmazione logica a vincoli

Concettualmente, l'assenza di tipi predefiniti comporta la necessità teorica di ridefinire in ogni programma logico i simboli e gli assiomi corrispondenti ai vari domini che abitualmente sono utilizzati nella risoluzione di pressoché tutti i problemi applicativi: interi, reali, ecc.. Nella pratica, tutti i sistemi di programmazione reali basati sulla logica forniscono tipi primitivi come interi e reali, ma con una visione funzionale che distorce il modello relazionale della programmazione logica.

I *linguaggi logici a vincoli*, così come definiti in [JaLa87], forniscono invece una soddisfacente caratterizzazione semantica dei domini predefiniti, quali reali, razionali, domini finiti, ecc..

Senza entrare in dettagli che non interessano in questa sede, ci limiteremo a osservare come estendere con i vincoli un linguaggio logico conduca a uno strumento praticamente più potente, anche per la maggiore dichiaratività da cui i suoi programmi sono necessariamente caratterizzati. È infatti possibile abbandonare l'approccio funzionale alla manipolazione dei tipi primitivi

proprio dei sistemi logici reali, per adottare un più consono stile dichiarativo.

D'altra parte, l'introduzione di tipi primitivi in un linguaggio logico significa dotare il sistema logico di maggiore intelligenza, e quindi demandare ancora di più il controllo al sistema stesso da parte del programmatore. Infatti, ciò si riflette in una strategia di ricerca che non è più semplicemente left-most, ma è assai più complessa, e necessita, per essere descritta opportunamente, di concetti come *constraint solver*, *store dei vincoli*, *consistenza dei vincoli*, ecc..

In generale, comunque, i linguaggi a vincoli promuovono un modello computazionale di tipo *data-driven* [Tre82]: vengono cioè eseguite quelle operazioni per le quali risultano disponibili dati sufficienti. Ciò ovviamente accade senza il controllo diretto del programmatore, che può pertanto disinteressarsi dell'ordine di esecuzione.

Poiché il modello a vincoli risponde ad alcune fondamentali esigenze poste dai linguaggi logici, il modello di riferimento per questo lavoro non è semplicemente quello del Prolog puro, ma questo stesso esteso con la capacità di gestire vincoli di vario tipo su domini fondamentali quali interi e reali. Questa, per altro, è anche la direzione intrapresa dai più diffusi strumenti commerciali [SICS94, Ecl92], a cui per altro questa tesi fa sempre programmaticamente riferimento.

1.2 Programmazione orientata agli oggetti

1.2.1 Cos'è la programmazione orientata agli oggetti?

Non si deve confondere l'essenza della programmazione a oggetti con la lista delle caratteristiche di questo o quel linguaggio a oggetti. Invece, si deve pensare il paradigma di programmazione *a oggetti* come un insieme di astrazioni e concetti indipendenti dal linguaggio ospite, come in [Weg87] (da cui derivano la maggior parte delle considerazioni che seguono). Da questo

punto di vista, il paradigma a oggetti può essere interpretato come un modello per la strutturazione dei programmi e l'organizzazione delle computazioni intorno agli elementi del dominio applicativo.

Ogni programma di un linguaggio a oggetti è strutturato intorno a un elemento del dominio applicativo. Sia il codice sia i dati relativi a un particolare oggetto sono raggruppati attorno a un unico elemento di programma, che definisce compiutamente l'oggetto stesso.

In fase d'esecuzione, ogni oggetto è un'entità indipendente che evolve inviando e ricevendo messaggi da altri oggetti. Una computazione a oggetti è quindi costituita da un insieme di entità indipendenti che scambiano messaggi tra loro. Lo stato della computazione consiste nell'insieme degli stati dei singoli oggetti, e il risultato stesso delle computazioni è espresso in termini degli oggetti generati/configurati dal programma nel corso della sua esecuzione.

Il paradigma di programmazione *orientato agli oggetti* può essere distinto da quello a oggetti (interpretando liberamente la classificazione di Wegner) per l'ulteriore capacità di specificare gli oggetti incrementalmente. Gli oggetti vengono definiti a partire da archetipi, detti *classi*, che possono essere associati in gerarchie classe/superclasse (classe/sottoclasse), in cui ogni sottoclasse *eredita* dalla superclasse comportamento e/o struttura. Ogni oggetto è poi un'*istanza* di una classe, e la sua struttura e il suo comportamento sono determinati dalla gerarchia di cui la sua classe fa parte. I meccanismi di ereditarietà possono mutare (e in generale mutano) da un linguaggio all'altro.

Vi sono poi una serie di caratteristiche tipiche del modello a oggetti, che ne fanno un formidabile strumento per lo sviluppo del software.

Innanzitutto, la capacità di costruire *astrazioni di dato*, nascondendo la rappresentazione dei dati nonché l'implementazione delle operazioni che su di essi possono essere compiute (*information hiding*). A tali astrazioni può essere associato un *sistema di tipi forte*, che impone vincoli statici

all'applicabilità delle operazioni.

L'identità degli oggetti, poi, può *persistere* da un'applicazione all'altra, indipendentemente dai valori o dalle chiavi utilizzate per accedere agli oggetti stessi.

Più in generale, le caratteristiche di un linguaggio orientato agli oggetti possono essere definite in un ideale spazio multidimensionale, le cui dimensioni sono costituite dalla capacità del singolo linguaggio di definire e realizzare concetti come *oggetto*, *classe*, *tipo*, *delegazione*, e *astrazione*. [Weg87] definisce tali concetti, e ne mostra l'*ortogonalità* e *consistenza* globale, ossia l'indipendenza reciproca e la possibilità di coesistenza in un unico linguaggio. Altre categorie ortogonali sono *concorrenza* e *persistenza*.

In breve, un *oggetto* è caratterizzato da un insieme di operazioni, e da uno stato, che tiene traccia degli effetti delle operazioni. Lo stato è dunque ciò che distingue un oggetto da una funzione: la sua reazione a un'operazione dipende dalla successione dei messaggi che l'oggetto scambia con gli altri oggetti. Ogni linguaggio che consenta di definire entità computazionali indipendenti di questo tipo è un linguaggio a oggetti.

Il *tipo* è ciò caratterizza ogni espressione del linguaggio che denoti un oggetto. I concetti di classe e di sistema di tipi forte possono essere visti come particolari specializzazioni del concetto di tipo.

D'altra parte, il meccanismo di *delegazione* è quello che consente a ogni oggetto di demandare la responsabilità di effettuare un'operazione o restituire un valore ad altri oggetti di una *gerarchia* (sia essa fissa o mutabile, sia essa determinata staticamente o dinamicamente) cui l'oggetto appartiene. Si vede come tale nozione comprenda in sè quella di *ereditarietà* come caso particolare, e che tutte le molteplici varietà di meccanismi che possono realizzare le diverse forme di ereditarietà implementate da vari linguaggi sono tutti riconducibili a diverse forme di delegazione.

L'*astrazione*, infine, è il meccanismo che consente la specifica delle interfacce. L'astrazione di dato è il principale uso del meccanismo di

astrazione in un linguaggio a oggetti.

1.2.2 Vantaggi della programmazione orientata agli oggetti

Il principale beneficio apportato da un approccio a oggetti nella progettazione e nello sviluppo del software è rappresentato dalla sua capacità di strutturare un programma secondo gli elementi del dominio applicativo. Ogni singolo oggetto del dominio applicativo può essere compiutamente descritto in maniera indipendente dagli altri, nascondendone i dettagli implementativi.

Inoltre, il comportamento di un oggetto è definito in termini del suo stato e dell'interazione con gli altri oggetti, cosicché una computazione di un linguaggio a oggetti può essere concepita come una rete di entità indipendenti e interconnesse che scambiano informazione tra loro tramite messaggi, e che evolvono in risposta all'informazione scambiata.

Tramite la possibilità di definire classi di oggetti, i linguaggi orientati agli oggetti consentono la condivisione del codice e il riuso dei componenti software. Tutti gli oggetti di una stessa classe condividono gli stessi metodi, e sono caratterizzati dalla stessa struttura.

La possibilità poi di definire gerarchie di classi, sfruttando meccanismi di ereditarietà, consentono un approccio incrementale alla costruzione del software. L'ereditarietà consente la definizione componenti software generici, che possono essere specializzati nel corso di successive fasi di sviluppo.

In generale, quindi, la metodologia di costruzione del software promossa dall'approccio a oggetti è tipicamente *bottom-up*: si parte con la definizione dei componenti software elementari, con cui possono poi essere combinati nella definizione di componenti via via più complessi e funzionalmente evoluti, sino a giungere alla costruzione di un sistema completo.

1.2.3 Limiti della programmazione orientata agli oggetti

Il primo limite dell'approccio a oggetti consiste nella difficoltà di applicare il procedimento di classificazione a comprendere la generalità di ogni dominio

applicativo. Creare tassonomie che siano in grado di comprendere tutte le specie di oggetti coinvolti in un problema applicativo raramente risulta un'operazione semplice.

In aggiunta, la maggior parte dei linguaggi a oggetti disponibili non consente la definizione di tassonomie se non staticamente. In tal modo, tutte le classi di oggetti devono essere definite direttamente dal programmatore, e non possono essere costruite come risultato della computazione.

Inoltre, i più diffusi linguaggi a oggetti sono linguaggi basati sull'assegnamento: la mancanza di trasparenza referenziale, nonché la possibilità di effetti collaterali, rendono usualmente difficile la lettura di un programma scritto in un linguaggio siffatto.

Tali linguaggi non consentono poi di trattare oggetti non completamente specificati: è tipico, infatti, che essi (p.e. il C++) forniscano la nozione di *costruttore* come particolare metodo che effettua l'operazione di inizializzazione di un oggetto, in modo che in nessun momento della computazione ci si trovi a calcolare su oggetti non completamente specificati.

In generale, poi, i linguaggi basati sull'assegnamento adottano un approccio non relazionale, e non consentono quindi di esprimere direttamente vincoli sullo stato di un oggetto: è così lasciata al programmatore la responsabilità del mantenimento della consistenza dello stato di ogni singolo oggetto in ogni istante della computazione, cosa sovente complicata dalla mancanza di strumenti per rendere atomica un'operazione di assegnamento non elementare.

I linguaggi a oggetti imperativi, in particolare, come il diffusissimo C++, adottano un modello computazionale tipicamente *control-driven*, che, se da una parte consente il massimo della flessibilità nella fase di stesura del programma, dall'altra costringe il programmatore a farsi carico *in toto* della parte di controllo di un algoritmo.

1.2.4 La nozione di stato nei linguaggi a oggetti

È proprio la prevalenza dei linguaggi imperativi tra i linguaggi a oggetti più diffusi a generare i più grossi equivoci sul concetto di stato in un linguaggio a oggetti. L'incapacità di gestire oggetti non completamente specificati porta a identificare due tipi di intervento sullo stato di un oggetto: l'*inizializzazione*, come prima fase, e la *modifica*, in tutte le fasi successive.

In realtà, come mostrato in [OmNa94], un'ovvia generalizzazione del concetto di stato di un oggetto porterebbe a identificare un'unica nozione a cui tutto può essere ricondotto: la *configurazione* dello stato di un oggetto.

Configurare un oggetto significa essenzialmente specificarne lo stato in maniera compiuta. L'operazione di inizializzazione dei linguaggi imperativi a oggetti è una configurazione completa che usualmente precede ogni altra operazione sull'oggetto. In effetti, in tutti i domini non dipendenti dal tempo, la ricerca delle corretta configurazione di un oggetto (come la ricerca delle soluzioni di un sistema di equazioni) non comporta concettualmente modifiche di stato.

In questo senso, la modifica dello stato di un oggetto è strettamente legata, almeno concettualmente, a domini dipendenti dal tempo. Modificare lo stato significa quindi *ri-configurare* l'oggetto stesso secondo i nuovi vincoli introdotti dallo scorrere del tempo, e dalla relativa evoluzione del dominio. Il cambiamento dello stato, quindi è ancora riconducibile, concettualmente, alla configurazione, anche se, a differenza dell'inizializzazione, non più effettuata necessariamente prima di ogni altra operazione sull'oggetto.

1.3 Sommario

Come la peculiarità del paradigma logico risiede innanzitutto nel suo *modello computazionale*, così quella del paradigma a oggetti risiede nel suo *modello dei dati*. Non esiste un modello computazionale che sia genericamente attribuibile alla programmazione a oggetti, e non solo a un particolare linguaggio a oggetti: il paradigma imperativo *non* è il luogo naturale

dell'approccio orientato agli oggetti. D'altra parte, la più grave carenza dei linguaggi logici risiede nell'assenza di un modello dei dati soddisfacente.

Esiste quindi una sorta di ortogonalità dei paradigmi logico e a oggetti, che rende concepibile una loro integrazione in un unico schema, che adotti un modello computazionale di tipo logico e un modello dei dati a oggetti. Costruire un linguaggio logico a oggetti, che adotti un approccio relazionale, che possegga una semantica dichiarativa, e che, nel contempo, fornisca quella capacità di modellazione dei domini applicativi tipica dei linguaggi a oggetti, è quindi l'obiettivo primario di questo lavoro.

2 Oggetti in programmazione logica

Il primo problema da affrontare nella definizione di un linguaggio orientato agli oggetti basato sulla logica del prim'ordine è *come rappresentare un oggetto*. Un programma orientato agli oggetti è strutturato secondo una partizione del dominio applicativo in elementi indipendenti. Gli oggetti del dominio sono classificati staticamente in tassonomie di classi. Una computazione a oggetti consiste in una collezione di entità indipendenti che comunicano tra loro scambiandosi messaggi. Ogni oggetto è dotato di un'identità propria, è denotato da un identificatore esplicito, e interagisce con gli altri oggetti secondo un determinato protocollo di comunicazione. Lo stato

di ogni oggetto, che incapsula lo stato complessivo della computazione, evolve in risposta all'informazione scambiata con gli altri oggetti. In particolare, ogni oggetto protegge il suo stato dall'esterno, cosicché la dinamica di una computazione a oggetti risulta dalla somma dei comportamenti esterni degli oggetti coinvolti, piuttosto che dall'unione dei loro stati.

Di conseguenza, qualunque scelta riguardo la rappresentazione di oggetti in programmazione logica deve consentire una definizione soddisfacente di nozioni come identità di un oggetto, stato di un oggetto, scambio di messaggi, ereditarietà, incapsulamento, information hiding, ecc.

2.1 Oggetti come termini

In logica, gli oggetti del dominio del discorso sono rappresentati per mezzo di termini. In questo senso risulta fondamentale la nozione di interpretazione di una teoria logica, o di un programma. Semplificando, l'interpretazione di un programma è una corrispondenza tra gli elementi dell'Universo di Herbrand del programma (che sono termini *ground*, ossia che non contengono variabili) e gli oggetti del dominio inteso.

Per questo, un programma logico viene di solito scritto per essere più un costruttore di termini che non un insieme di assiomi da cui derivare teoremi, con ciò mettendo l'accento più sulla nozione di dimostrazione costruttiva piuttosto che sulla dinamica di successo e fallimento di una computazione logica.

Esempio 2.1

Prendiamo ad esempio una classe di semplici resistori, il cui unico requisito sia quello di soddisfare la legge di Ohm. Ogni resistore è rappresentato da un termine `resistor/3`, i cui argomenti definiscono nell'ordine la caduta di potenziale ai morsetti, il valore di resistenza, e la corrente che attraversa il componente resistivo.

Adottando la convenzione sintattica per la scrittura delle procedure per cui il primo argomento di ogni metodo rappresenta il destinatario del messaggio, una classe di resistori (intesa come collezione di proprietà comuni) può essere rappresentata dalla seguente teoria:[2]

```
ohm(resistor(V, R, I)) :- V = R * I.                    %a1

voltage(resistor(V, _, _), V).                          %a2

value(resistor(_, R, _), R).                            %a3

current(resistor(_, _, I), I).                          %a4

equivalent(R, series(R1, R2)) :-                        %a5
      voltage(R, V), voltage(R1, V1), voltage(R2, V2),
      current(R, I), current(R1, I), current(R2, I),
      V = V1 + V2.

equivalent(R, parallel(R1, R2)) :-                      %a6
      voltage(R, V), voltage(R1, V), voltage(R2, V),
      current(R, I), current(R1, I1), current(R2, I2),
      I = I1 + I2.
```

Queste clausole possono essere usate per controllare semplicemente la consistenza di una data triple di valori costituenti lo stato di un resistore:

```
?-  ohm(resistor(10,5,2)).                             %g1
yes

?-  ohm(resistor(5,10,2)).                             %g2
no
```

Allo stesso modo, questo programma può essere usato per costruire resistori che soddisfino determinate specifiche:

```
?-  current(R1,3), value(R1,4), ohm(R1).               %g3
R1 = resistor(12,4,3) ?
```

Metodi come `voltage/2` possono essere utilizzati sia come selettori, sia come costruttori, sfruttando le proprietà dell'unificazione come metodo uniforme

---

[2] D'ora in poi, secondo l'idea che il "linguaggio contenitore" non è il semplice Prolog standard, ma una sua versione estesa ai vincoli, utilizzeremo liberamente notazioni e soluzioni tipiche di linguaggi logici a vicoli. Nel corso di tutto questo capitolo, per esempio, supporremo di avere a disposizione un CLP(*R*) in grado tra le altre cose di risolvere le equazioni risultanti dall'applicazione della legge di Ohm.

per passaggio, selezione e costruzione di dati.

```
?-  R1 = resistor(12,4,3), voltage(R1, V).          %g4
R1 = resistor(12,4,3), V = 12 ?

?-  R1 = resistor(V,4,3), voltage(R1, 12).          %g4'
R1 = resistor(12,4,3), V = 12 ?

?-  R1 = resistor(V,4,3), voltage(R1, V').          %g4"
R1 = resistor(V,4,3), V = V' ?
```

Tuttavia, ogni metodo relativo ai resistori ne sfrutta la rappresentazione esplicita come termine, consentendo quindi l'accesso al suo stato senza nessuna possibile protezione. Inoltre, non c'è nessuna garanzia che un oggetto-resistore costruito da `voltage/2` (o da altri metodi) sia corretto, ossia abbia uno stato consistente. Infatti, hanno successo pure i seguenti goal, nonostante le configurazioni dei resistori risultanti non abbiano senso:

```
?-  R1 = resistor(10,10,10), voltage(R1, V).        %g5
R1 = resistor(10,10,10), V = 10 ?

?-  R1 = resistor(V,10,10), voltage(R1, 10).        %g5'
R1 = resistor(10,10,10), V = 10 ?

?-  R = resistor(10,20,I), R1 = resistor(10,40,15),  %g6
    current(R2,15), equivalent(R, parallel(R1, R2)).
I = 30,
R = resistor(10,20,30),
R1 = resistor(10,40,15),
R2 = resistor(10,_,15) ?
```

Un'approccio più plausibile può quindi essere ottenuto introducendo un unico metodo "creatore" (che, a sua volta, può poi essere utilizzato sia come costrutture, sia come selettore, sia per la verifica della consistenza), come il seguente `isaResistor/4`,

```
isaResistor(resistor(V, R, I), V, R, I) :-          %a0
    V = R * I.
```

e forzandone l'uso ogni volta è necessario introdurre un nuovo resistore, come nel caso della seguente riscrittura dell'assioma (`a5`):

```
equivalent(R, series(R1, R2)) :-                    %a5'
      isaResistor(R, V, _, I),
      isaResistor(R1, V1, _, I),
      isaResistor(R2, V2, _, I),
      V = V1 + V2.
```

Riscrivendo tutti i metodi secondo questa strategia, i goal (`g5-6`) falliscono, ed è possibile effettuare computazioni più complesse come le seguenti:

```
?-  current(R1, 3), isaResistor(R1, _, 4, I),       %g7
    isaResistor(R2, _, 2, I).
I = 3,
R1 = resistor(12,4,3),
R2 = resistor(6,2,3) ?

?-  isaResistor(R1, 12, 4, _), isaResistor(R2, _, 4, _),
    equivalent(R, parallel(R1,R2)).                 %g8
R1 = resistor(12,4,3),
R2 = resistor(12,4,3),
R = resistor(12,2,6) ?
```

Il concetto di *costruzione di oggetti* è quindi assai rilevante in programmazione logica. Da una parte, la costruzione di oggetti (come configurazione di termini) è un tipico risultato inteso di una computazione logica (come emerge in particolare dai goal (`g3`), (`g7`), e (`g8`)). Dall'altra, poiché tengono traccia dell'evoluzione della computazione, l'opzione di rappresentare gli oggetti come termini consente di catturare la nozione di stato di un oggetto, seppure sotto certi limiti.

Rispetto ai più diffusi linguaggi a oggetti, basati sul paradigma imperativo, l'approccio logico offre diversi vantaggi, come l'invertibilità dei parametri, e la capacità di esprimere vincoli intra- e inter-oggetto come relazioni tra termini.

Questa stessa scelta, tuttavia, si risolve in un approccio molto debole, non consentendo né l'incapsulamento né l'astrazione dei dati, e non fornendo alcun mezzo per esprimere le nozioni di classe, gerarchie di classi, ed

ereditarietà.[3] In particolare, la questione dell'identità di un oggetto rimane irrisolta, non sussistendo distinzione tra l'oggetto e il suo termine identificatore: due oggetti distinti non possono avere lo stesso stato (altrimenti risulterebbero tra loro indistinguibili), e nessun oggetto può modificare il suo stato.

2.2 Oggetti come collezioni di clausole

La difficoltà di rappresentare e manipolare oggetti complessi tramite termini forza sovente il programmatore logico ad adottare una rappresentazione a clausole invece che a termini. L'idea fondamentale, formulata in [McC92], è che un'oggetto è ciò che noi sappiamo essere vero di esso. Quindi, le clausole di un programma devono essere in qualche modo "etichettate", in modo da determinarne la corrispondenza rispetto agli oggetti del dominio applicativo.

Una prima, banale tecnica è quella di usare il primo argomenti di ogni atomo nelle clausole di un programma allo scopo di identificare l'oggetto inteso, denotato da un particolare termine ground.

Esempio 2.2

Possiamo quindi pensare di rappresentare ogni resistore come un insieme di clausole. Se per esempio `r1` è il resistore con `V=12`, `R=4` e `I=3`, scriveremo semplicemente qualcosa come

```
state(r1,12,4,3).                                    %s1
```

oppure, equivalentemente,

```
voltage(r1,12).                                      %s2
value(r1,4).                                         %s3
current(r1,3).                                       %s4
```

Solo alcune delle clausole dell'Esempio 2.1, rappresentanti delle specie di

---

[3] Un possibile approccio a questo problema si può ritrovare in [AiNa86].

metodi della classe dei resistori, devono essere riscritti. Le clausole (`a0-1`) devono essere modificate, dato che accedono direttamente alla rappresentazione dello stato, mentre (`a5-6`) possono essere lasciate immodificate.

```
isaResistor(Res, V, R, I) :-                          %a0
    voltage(Res, V), value(Res, R), current(Res, I),
    V = R * I.
ohm(Res) :-                                           %a1
    voltage(Res, V), value(Res, R), current(Res, I),
    V = R * I.
```

Questa riscrittura mostra anche può non essere necessario avere un corrispettivo delle clausole (`a2-4`), una volta che si sia scelta un'opportuna rappresentazione dello stato (in questo caso, la soluzione (`s2-4`)). Altrimenti, con la soluzione (`s1`), avremmo semplicemente scritto:

```
voltage(Res, V) :- state(Res, V, _, _).               %a2
value(Res) :- state(Res, _, R, _).                    %a3
current(Res) :- state(Res, _, _, I).                  %a4
```

Inutile a dirsi, anche questo approccio risulta molto debole, dato che tutto, dall'identità di un oggetto alla relazione classe/istanza, è lasciato all'interpretazione del programmatore. Il problema risiede in particolare nel ruolo dell'identificatore. Assegnare tale ruolo al primo argomento di ogni predicato (come nell'esempio 2.2) è una scelta del tutto arbitraria che non introduce nessuna distinzione reale tra l'identificatore del destinatario e gli argomenti del messaggio. Al contrario, il loro diverso ruolo a livello semantico dovrebbe ragionevolmente avere una corrispondenza a livello sintattico.

2.2.1 Oggetti come teorie logiche aperte (etichettate)

L'idea di base è di partizionare lo spazio degli assiomi di un programma logico in teorie separate, ciascuna delle quali descrive un particolare elemento

del dominio inteso. Il framework logico *multiteoria* (o *a teorie multiple*) risultante corrisponde bene alla nozione di suddivisione del dominio applicativo in entità indipendenti. Ivi, ciascun oggetto può essere descritto tramite la collezione degli assiomi che definiscono ciò che è vero rispetto a esso.

Un ambito logico multiteoria richiede una diversa nozione di verità per una formula. Allo stesso modo in cui un'assioma non fa più riferimento a verità universali, ma solo locali a una singola entità, più in generale una formula logica può essere detta vera solo rispetto a una particolare astrazione del dominio del discorso. In generale, un framework multiteoria introduce nella programmazione logica la questione del *contesto di prova*. Dato un programma *P* e una formula atomica *G*, chiedere se *G* può essere derivato da *P* non ha più senso; invece, se *O* è una teoria di *P*, potremmo cercare di derivare *G* da *O*, adottando cioè *O* come contesto di prova per *G*. Di qui in avanti, denoteremo con *O*:*G* la *formula etichettata* che è vera quando *G* è derivabile dalla teoria *O* (o, più in generale, dal contesto di prova *O*).

Adottando un'ottica a oggetti, il fatto che ogni formula debba essere associata a un particolare contesto di prova induce una sorta di protocollo a scambio di messaggi per la dimostrazione di un goal: possiamo leggere *O*:*G* come l'invio del messaggio *G* all'oggetto *O*.[4] Per rafforzare tale interpretazione, adotteremo nel seguito negli esempi la sintassi `O<-G` per la formula logica *O*:*G*, anche se, da un punto di vista strettamente logico, proprio la sintassi opposta (`O->G`) sarebbe stata la più adatta (poiché tale formula risulta vera se *G* segue logicamente da *O*).

La seconda conseguenza fondamentale dell'adozione di un approccio multiteoria è che non sussiste più l'idea che una teoria logica contenga la conoscenza completa sul mondo, giacché essa rappresenta solo una descrizione incompleta di una parte del dominio. In effetti, la dimostrazione di

---

4 Per maggiori dettagli sui fondamenti teorici, si consulti l'appendice B.

una formula rispetto a una teoria può dipendere dalla conoscenza contenuta in altre teorie. In particolare, una formula atomica *G* può essere vera in *O* (ossia, vale *O*:*G*) una volta che sia data la verità di O_1:G_1, O_2:G_2, …, O_n:G_n: cioè, al fine di dimostrare *G*, la teoria *O delega* la prova dei goal G_i alle teorie O_i. Per questo, un ambiente multiteoria consiste necessariamente di *teorie logiche aperte* [BLM91].

Infine, per denotare gli oggetti si devono introdurre gli *identificatori di teoria*. Allo scopo di mantenere il tipico approccio logico, gli oggetti dovrebbero essere ancora denotati per mezzo di termini. Scegliamo perciò di utilizzare *termini ground* per denotare teorie logiche, piuttosto che usare nomi extra-logici (come in [BLM92a][5]). Nel nostro framework, quindi, un programma è costituito da una molteplicità di *teorie etichettate*, ciascuna delle quali denotata da un termine ground che ne rappresenta l'etichetta. Assegnare un nome a una teoria logica corrisponde a stabilire un link di metalivello tra ciascun termine ground che sia un'etichetta e la collezione di clausole logiche che ne costituiscano la corrispondente teoria etichettata. Così, in un programma logico a teorie multiple etichettate, un'interpretazione associa a ogni oggetto del dominio, tramite un identificatore di teoria, un insieme di assiomi che definiscono le proprietà dell'oggetto stesso.

Il presente approccio risolve il problema dell'identità di un oggetto, giacché la denotazione dell'oggetto nel programma è ben distinta dalla sua rappresentazione concreta (ossia la rappresentazione delle sue proprietà). Metodi e attributi dello stesso oggetto possono ora essere incapsulati nella stessa struttura (la teoria logica corrispondente), con l'ulteriore beneficio derivante dall'uniforme rappresentazione in forma di clausole. È possibile a

---

5 Pag. 108: "The names of the objects are not included in H_P". Questo ha almeno due conseguenze negative, ragione per cui si è qui scelto l'approccio opposto. In primo luogo, non è concettualmente possibile stabilire relazioni tra oggetti tramite i loro identificatori (come "*partOf*(*Id1*, *Id2*)"). In secondo luogo, viene tradito il tipico approccio della programmazione logica: gli oggetti del dominio inteso non sono più tutti rappresentati nell'Uiverso di Herbrand del programma, giacché alcuni di essi sono invece denotati tramite identificatori extra-logici.

questo punto definire un modello per l'information hiding, non essendovi più legami a priori tra la denotazione di un oggetto e la conoscenza che contiene.

Esempio 2.3

Questo esempio riprende gli esempi precedenti, seguendo stavolta un approccio a teoria multiple con etichetta. Non si è voluto per ora definire una sintassi di riferimento (cosa che verrà fatta nei capitoli successivi), adottando dei "contenitori-teoria" generici, come nel codice che segue:

theory `resistor`

```
isaResistor(Res, V, R, I) :-                          %a0
      Res <- (voltage(V), value(R), current(I)),
      V = R * I.

equivalent(ResEq, series(Res1, Res2)) :-              %a5
      ResEq <- (voltage(V), current(I)),
      Res1 <- (voltage(V1), current(I)),
      Res2 <- (voltage(V2), current(I)),
      V = V1 + V2.

equivalent(ResEq, parallel(Res1,Res2)) :-             %a6
      ResEq <- (voltage(V), current(I)),
      Res1 <- (voltage(V), current(I1)),
      Res2 <- (voltage(V), current(I2)),
      I = I1 + I2.
```

theory `r1`

```
voltage(12).                                          %s2
value(4).                                             %s3
current(3).                                           %s4
```

theory `r2`

```
voltage(12).                                          %s2'
value(2).                                             %s3'
current(6).                                           %s4'
```

I seguenti goal di successo rappresentano rispettivamente l'accesso allo stato di due resistori, e la verifica di una semplice equivalenza circuitale:

```
?-  resistor <- isaResistor(r1, V, R1, I1),           %g9
    resistor <- isaResistor(r2, V, R2, I2).
I1 = 3, I2 = 6, R1 = 4, R2 = 2, V = 12 ?
```

```
?-  resistor <- equivalent(r2, parallel(r1, r1)).    %g10
yes
```

La delegazione esplicita tramite invio di messaggio non rende conto di una forma di delegazione implicita che è fondamentale in programmazione a oggetti: l'ereditarietà. È quindi necessario individuare un modo per definire gerarchie di classi e per sfruttare meccanismi di delegazione diversi. Estendiamo quindi il presente framework a catturare relazioni classe/istanza, nonché classe/superclasse.

2.2.2 Oggetti come composizione di teorie logiche aperte

L'estensione del framework multiteoria con un semplice meccanismo di composizione di teorie consente l'introduzione di classi ed ereditarietà in ambito logico. In letteratura sono disponibili diverse possibilità per la composizione di teorie logiche: le "teorie logiche aperte" [BLM91], le teorie Ω-open di [BGLM92], la programmazione logica contestuale [MoPo89, MeNa92, DNO92], e altre [Bug92].

Tutte le proposte hanno in comune l'idea che le teorie logiche possono essere concepite come componenti software aperti che possono essere combinati a formare componenti più complessi. Il risultato della composizione di teorie logiche aperte può essere utilizzato come contesto di prova per formule atomiche. Le differenze tra i diversi approcci risiedono invece nel grado di "apertura" delle teorie, e nei meccanismi di composizione.

Prendendo a esempio un'estensione (come definita in [CSM]) del modello della programmazione logica contestuale, mostriamo ora come un ambiente logico con composizione di teorie consente di catturare nozioni come classi ed ereditarietà. In breve[6], le teorie logiche aperte sono dette *unità*, e possono essere combinate sia staticamente, sia dinamicamente, in sequenze

---

6 Una descrizione dettagliata del modello contestuale esteso si trova nel capitolo successivo.

dette *contesti*, che possono essere denotati come liste di unità. I contesti possono essere utilizzati per rappresentare tassonomie di oggetti, riproducendo la dinamica dei rapporti classe/istanza e classe/superclasse. In particolare, le tassonomie di teorie possono essere lette come gerarchie *is-a*. Un contesto $[T, S_k, \ldots, S_1]$ può essere interpretato come un'istanza T della classe S_k (che a sua volta ha $S_{k-1}, \ldots, S_1$ come superclassi), oppure, allo stesso modo, come una classe T che specializza una super classe S_k. Infine, le politiche di *binding eager* e *late* possono essere usate per riprodurre i meccanismi di binding *super* e *self* tipici dei linguaggi a oggetti: corrispondentemente, l'esempio sotto sfrutta gli operatori `xuper` e `xelf` per le politiche eager e late, rispettivamente.

Esempio 2.4

unit `resistor`

```
isaResistor(V, R, I) :-                                    %a0
    xelf (voltage(V), value(R), current(I)),
    V = R * I.

ohm :-                                                     %a1
    xelf (voltage(V), value(R), current(I)),
    V = R * I.

equivalent(series(Res1, Res2)) :-                          %a5
    xelf (voltage(V), current(I)),
    Res1 <- (voltage(V1), current(I)),
    Res2 <- (voltage(V2), current(I)),
    V = V1 + V2.

equivalent(parallel(Res1,Res2)) :-                         %a6
    xelf (voltage(V), current(I)),
    Res1 <- (voltage(V), current(I1)),
    Res2 <- (voltage(V), current(I2)),
    I = I1 + I2.
```

unit `colouredResistor`

```
isaColouredResistor(C, V, R, I) :-                         %a7
      xuper resistor(V, R, I),
      xelf colour(C).
```

unit `r1`

```
voltage(12).                                          %s2
value(4).                                             %s3
current(3).                                           %s4
```

unit r2

```
voltage(12).                                          %s5
value(2).                                             %s6
current(6).                                           %s7
colour(green).                                        %s8
```

Il resistore `r1`, istanza della classe `resistor`, può essere rappresentato tramite il contesto `[r1,resistor]`. Di conseguenza, i goal (`g9-10`) dell'esempio 2.3 possono essere riscritti alla seguente maniera:

```
?- [r1,resistor] <- isaResistor(V,R1,I1),             %g9
   [r2,colouredResistor,resistor] <-
                  isaColouredResistor(C,V,R2,I2).
C = green, I1 = 3, I2 = 6, R1 = 4, R2 = 2, V = 12 ?

?- R1 = [r1,resistor], R2 = [r2,resistor],
   R2 <- equivalent(parallel(R1,R1)).                 %g10
R1 = [r1,resistor], R2 = [r2,resistor] ?
```

In termini di "dimensioni" dello spazio di definizione dei linguaggi a oggetti, come descritto in [Weg87], un linguaggio logico basato sulla composizione di teorie offre diverse caratteristiche rilevanti.

Un oggetto può essere rappresentato da una collezione di teorie logiche in cui operazioni differenti condividono dati comuni. I dati possono essere incapsulati in una teoria "istanza", come fatti che rappresentano gli attributi di un oggetto. Le operazioni comuni a una classe di oggetti possono essere fattorizzate in teorie "classe". Teorie organizzate in gerarchie possono ereditare proprietà sia dinamicamente, sia staticamente da altre teorie. Il livello di granularità dei predicati può servire a costruire un filtro sulla visibilità delle clausole (e dei fatti, in particolare) al di fuori di un oggetto, fornendo così un meccanismo per l'information hiding e la costruzione di astrazioni di dato. Inoltre, la rappresentazione a clausole di Horn di tutte le proprietà consente l'adozione delle stesse politiche per il binding di metodi e attributi. Infine, la

rappresentazione a clausole consente la persistenza degli oggetti da una computazione all'altra, rafforzando così l'interpretazione di base di dati dei programmi logici.

D'altra parte questo approccio soffre di molteplici, severe limitazioni. In breve, le teorie "classi" non sono davvero classi, le "istanze" non sono vere istanze con uno stato proprio, e la nozione stessa di oggetto è riprodotta in modo assai povero.

Le teorie logiche usate come classi hanno metodi che condividono fatti costituenti attributi, ma non si può dire che fungano da template per un'istanza. Inoltre, nulla tranne l'interpretazione del programmatore consente di distinguere tra classi e istanze, essendo queste comunque teorie logiche dello stesso genere. Perciò, siccome le teorie che fungono da istanze possono solamente essere dichiarate in modo statico nel testo di un programma, e non possono essere costruite come risultato di una computazione guidata da altre teorie, potrebbero in linea di principio risultare strutturalmente difettose (ossia, contenere clausole non corrispondenti agli attributi, o mancare di fatti necessari alla rappresentazione dell'oggetto), o, peggio, essere corrette strutturalmente, ma risultare inconsistenti rispetto alle specifiche della classe.

Esempio 2.5

Le teorie `r11` e `r22` che seguono potrebbero essere utilizzate come "istanze" per la "classe" `resistor`, anche entrambe sono strutturalmente difettose, e `r22` è pure non coerente con le specifiche della classe dei resistori.

```
unit r11

state(10, 5, 2).

unit r22

voltage(10).
value(4).
current(3).
madeof(silicon).
```

In tal modo, nessuno dei sottogoal seguenti ha successo:

```
?- [r11,resistor] <- isaResistor(V,R,I).              %g9'
no

?- [r22,resistor] <- isaResistor(V,R,I).              %g9"
no
```

Non è possibile la creazione dinamica degli oggetti, che devono invece essere definiti tutti staticamente nel programma. I loro attributi non possono essere configurati come risultato di una computazione. In effetti, questo approccio non consente affatto di catturare il concetto di stato di un oggetto come memoria di una computazione.

Gli "oggetti" qui non sono altro che una particolare organizzazione del programma, ben diversi quindi dall'essere quelle unità atomiche d'incapsulamento dello stato di una computazione che un modello computazionale richiede per poter essere detto a oggetti [Weg90,Weg92b]. Al contrario, lo stato (e il risultato di una computazione) è ancora espresso in forma di termini, in modo scorrelato dalle definizioni degli oggetti, invece che in termini di configurazione di oggetti.

Il protocollo di scambio dei messaggi è pienamente direzionale, in contrasto con la tipica interpretazione "neutra" dei linguaggi relazionali: una (meta-)relazione "demo" non è invertibile. Infatti, il destinatario di un messaggio dev'essere completamente configurato per potere rispondere, e non può essere costruito come risultato di una sequenza di messaggi scambiati durante una computazione.

In definitiva, questo approccio disattende fondamentalmente sia la filosofia della programmazione logica, sia quella della programmazione a oggetti.

2.2.3 Oggetti come teorie parametriche

Un approccio più soddisfacente è quello di rappresentare gli oggetti come

(composizione di) teorie parametriche (come in [McC92][7] e, per certi versi, [Zan84]). Le teorie parametriche sono collezioni di assiomi che possono avere variabili libere. Queste ultime si considerano quantificate universalmente solo a livello di teoria. Queste variabili fungono da variabili d'istanza globali, che possono venire referenziate per nome in tutte le formule della teoria. Quindi, ciascuna teoria può essere caratterizzata attraverso un termine identificatore, il cui funtore è il nome atomico di una teoria parametrica e i cui argomenti sono le variabili globali. Ogni teoria parametrica `c` può essere vista come una classe, poiché rappresenta l'insieme di tutti i suoi oggetti-istanza: il suo termine identificatore $\mathtt{c}(\tilde{X})$ (in cui $\tilde{X}$ rappresenta una tuple di variabili) può essere interpretato come un *template* per tutte le istanze di `c`, ciascuna delle quali è a sua volta rappresentata da una sostituzione ground delle variabili $\tilde{X}$. Pertanto useremo nel seguito il termine *classe* per riferirci a tali template (corrispondenti ai Class Templates di McCabe), e il termine *istanza* per le teorie ottenute istanziando le variabili globali di una classe.

Esempio 2.6

Una teoria parametrica `resistor` può essere rappresentata da una formula chiusa come la seguente:

$$\mathtt{resistor}: \forall v, r, i \begin{cases} \mathtt{ohm} \leftarrow =(v, *(r, i)) \\ \mathtt{voltage}(v) \leftarrow \\ \mathtt{value}(r) \leftarrow \\ \mathtt{current}(i) \leftarrow \end{cases}$$

che potrebbe essere codificata nel modo seguente:

```
theory resistor(V,R,I)
ohm :- V = R * I.                    %a1
voltage(V).                          %a2
value(R).                            %a3
```

[7] Finché possibile, cercheremo di astrarre dalle scelte di McCabe per discutere potenzialità e problemi teorici dell'approccio in generale.

```
current(I).                                          %a4
```

Si può vedere come la teoria `resistor(V,R,I)` sia un template per le teorie istanza di `resistor`. Ciascuna istanza ground di `resistor(V,R,I)` rappresenta un programma logico standard ottenuto sostituendo opportunamente gli argomenti ground alle variabili parametriche della teoria. Per esempio, il termine ground `resistor(10,2,5)` rappresenta un'istanza della classe `resistor` costituita dalle seguenti clausole di Horn:

theory `resistor(10,2,5)`

```
ohm :- 10 = 2 * 5.                                   %a1'
voltage(10).                                         %a2'
value(2).                                            %a3'
current(5).                                          %a4'
```

Query come le seguenti

```
?-  resistor(10,2,5) <- ohm.
yes
?-  resistor(10,2,5) <- voltage(V).
V = 10 ?
```

mostrano possibili messaggi inviati alla teoria `resistor(10,2,5)`, insieme alle risposte corrispondenti.

Le teorie parametriche consentono un interessante approccio alla reppresentazione dei concetti di classe e istanza in ambito logico. Una classe denota l'insieme (eventualmente infinito) delle sue istanze, e ogni oggetto è associato all'insieme dei metodi di classe corrispondenti, che condividono l'informazione di stato. In aggiunta, "aprendo" le teorie parametriche, e consentendone la composizione (come visto per le teorie definite nella sottosezione precedente) è possibile definire tassonomie di oggetti, e una classe può ereditare metodi dalle sue superclassi. Naturalmente, è necessario introdurre un nuovo meccanismo per la condivisione dei dati di un'istanza, poiché la specificazione delle variabili che costituiscono gli attributi cambia

dalla classe alla superclasse, non consentendo, in generale, agli attributi di essere referenziati uniformemente lungo tutta una gerarchia di teorie. In tal modo, ogni variabile globale di una superclasse può essere istanziata secondo quanto specificato nella sottoclasse, ed è possibile accedere ai valori degli attributi direttamente da tutta la gerarchia.

Tuttavia, risulta necessario effettuare preliminarmente una distinzione fondamentale: una teoria parametrica è semplicemente un template per una collezione di clausole di Horn oppure è una teoria logica a tutti gli effetti, costituita di formule che potremmo dire "clausole di Horn estese"? La prima scelta (che diremo *opzione dell'istanza ground*) corrisponde ad accettare solo termini ground come identificatori di teorie logiche, essendo questi gli unici in grado di rappresentare programmi logici definiti. La seconda scelta (che diremo *opzione dell'istanza non ground*) consentirebbe invece di delegare la prova di un goal anche a teorie non completamente specificate, forzando però l'utilizzo di una procedura di dimostrazione non standard per formule che non sono clausole di Horn.

L'opzione dell'istanza ground limita in modo severo l'utilità dell'approccio in discussione. Ancora una volta, il protocollo di scambio di messaggi sarebbe in questo caso pienamente direzionale, e nessuna istanza potrebbe essere configurata in risposta a un messaggio ricevuto. In tal modo, la configurazione di un oggetto in stile pienamente relazionale risulterebbe impossibile, mentre proprio questo è uno dei requisiti fondamentali di questo lavoro. Inoltre, lo stato di un oggetto non può in alcun modo incapsulare una porzione dello stato globale della computazione, non realizzando così un altro dei requisiti fondamentali.

Dimenticando per il momento la necessità di introdurre una procedura di prova non standard,[8] l'opzione dell'istanza non ground rappresenta

---

[8] Reinterpretando i suggerimenti di [McC92] nel paragrafo "The semantics of variable labels", si dovrebbe pensare alle variabili di clausole non locali come se esse fossero potenzialmente ground, e non fossero che "segnaposto" per configurazioni di stato a

chiaramente l'approccio migliore. La nozione di oggetto è qui adeguatamente riprodotta. Non solo, infatti, ogni oggetto risulta associato a un insieme di metodi che condividono informazioni di stato comuni, ma gli attributi di un oggetto costituiscono effettivamente una porzione incapsulata dello stato globale della computazione. È così possibile configurare lo stato di un'istanza incrementalmente, con l'unificazione, in modo da costruire un oggetto come risultato di una dimostrzione logica. Il protocollo di scambio di messaggi è neutrale, dato che l'oggetto destinatario può cambiare la sua configurazione in risposta al messaggio ricevuto, promuovendo così uno stile pienamente relazionale.

Esempio 2.7

Un primo tentativo potrebbe portare alla seguente riscrittura degli assiomi della teoria `resistor`:

theory `resistor(V,R,I)`

```
ohm :- V = R * I.                                          %a1
voltage(V).                                                %a2
value(R).                                                  %a3
current(I).                                                %a4
equivalent(series(Res1, Res2)) :-                          %a5
      Res1<- (voltage(V1), current(I)),
      Res2<- (voltage(V2), current(I)),
      V = V1 + V2.
equivalent(parallel(Res1,Res2)) :-                         %a6
      Res1<- (voltage(V), current(I1)),
      Res2<- (voltage(V), current(I2)),
      I = I1 + I2.
```

Sin qui, come nell'esempio 2.1, risulta ancora possibile dimostrare formule "sbagliate" come le seguenti:

---

venire. Tali clausole sarebbero quindi da intendersi come clausole di Horn "in fieri", e dovrebbero essere pertanto trattate come tali.

```
?-  resistor(10,10,10) <- voltage(V).                %g5
V = 10 ?

?-  resistor(10,20,I) <-                             %g6
    equivalent(
      parallel(resistor(10,40,15), resistor(10,40,15))
    ).
I = 30 ?
```

Gli assiomi (`a2-6`) necessitano quindi di un'ulteriore riscrittura, analogamente a quanto fatto nell'esempio 2.1, per introdurre la verifica di consistenza in ogni metodo:

```
voltage(V) :- ohm.                                   %a2

value(R) :- ohm.                                     %a3

current(I) :- ohm.                                   %a4

equivalent(series(Res1, Res2)) :-                    %a5
      ohm,
      Res1<- (voltage(V1), current(I)),
      Res2<- (voltage(V2), current(I)),
      V = V1 + V2.

equivalent(parallel(Res1,Res2)) :-                   %a6
      ohm,
      Res1<- (voltage(V), current(I1)),
      Res2<- (voltage(V), current(I2)),
      I = I1 + I2.
```

La prova dei goal (`g5-6`) ora fallisce, ed è possibile realizzare computazioni come le seguenti:

```
?-  R1 = resistor(_, 4, I) , R1 <- current(3),       %g7
    R2 = resistor( _, 2, _), R2 <- current(I).
I = 3,
R1 = resistor(12, 4, 3), R2 = resistor(6, 2, 3) ?

?-  R1 = resistor(12, R, _), R1 <- value(4),         %g8
    R2 = resistor(_, R, _), Rpar = resistor(_, _, _),
    Rpar <- equivalent(parallel(R1,R2)).
R = 4, R1 = resistor(12, 4, 3),
R2 = resistor(12, 4, 3), Rpar = resistor(12, 2, 6) ?
```

La teoria parametrica seguente consente d'esemplificare i meccanismi d'ereditarietà in questo contesto:

```
theory_colouredResistor(C,V,R,I)
colour(C) :- ohm.                                          %a7
```

Per vincolare le teorie `resistor` e `colouredResistor` in una relazione classe/superclasse, e condividere gli attributi della superclasse, si può pensare per esempio di consentire l'asserzione di un assioma come quello qui sotto:

```
colouredResistor(C,V,R,I) ∝ resistor(V,R,I)                %m1
```

che leghi le due teorie come classe e superclasse, e ne correli al contempo le variabili-attributo.

Ciò porta al successo le computazioni seguenti:

```
?-  Rp = resistor(V, _, 3), Rp <- value(4),                %g11
    Re = coloured_resistor(green, V, _, _),
    Re <- equivalent(parallel(Rp,Rp).
Re = coloured_resistor(green,12,4,3),
Rp = resistor(12,4,3), V = 12 ?
```

Tuttavia, anche questo approccio soffre di diverse, non trascurabili limitazioni.

In primo luogo, non è possibile utilizzare le teorie parametriche come classi a pieno titolo. Poiché costuiscono a tutti gli effetti template per le loro istanze, le teorie parametriche possono essere utilizzate per configurare un oggetto; tuttavia, non servono a crearlo (si vedano i goal `g7`, `g8` e `g11` nell'esempio 2.7). Un oggetto può essere costruito in maniera inconsistente rispetto alle proprietà di una data classe, ed essere usato ugualmente come membro della classe stessa.

Inoltre, l'ereditarietà riguarda solo le proprietà procedurali, ossia i metodi: gli attributi non possono essere ereditati direttamente da una teoria all'altra, ma devono essere legati esplicitamente tramite un'opportuna dichiarazione (come la dichiarazione `m1` della teoria `coloured_resistor` nell'esempio 2.7).

La denotazione di un oggetto è qui di nuovo un problema, poiché, come

nel caso degli oggetti come termini (sezione 2.1), essa coincide con la configurazione dell'oggetto. Ciò preclude l'indipendenza dalla rappresentazione, la protezione dello stato, e la costruzione di astrazioni di dato. Per ragioni analoghe, non è possibile esprimere la nozione di stato modificabile, dato che lo stato di un oggetto coincide con la sua denotazione.

Una soluzione insoddisfacente a questo problema contempla la possibilità di associare identificatori atomici (degli *alias*) a termini-istanza (come nelle *class rules* di McCabe), separando così la denotazione dell'oggetto dalla sua rappresentazione concreta, e consentendo l'incapsulamento e la protezione dell'informazione di stato. Questo, peraltro, sarebbe un rimedio peggiore del male: prima di tutto, nulla cambierebbe per ciò che riguarda le limitazioni su classi ed ereditarietà di proprietà. Per esempio, si potrebbero ancora creare "istanze" senza alcuna relazione con la specifica della classe. Inoltre, ciò creerebbe nuovi problemi: ogni alias dovrebbe essere necessariamente associato a un termine ground di un'istanza per essere significativo (associandola a un termine contenente variabili denoterebbe in generale, e permanentemente, ancora una classe di oggetti, e non una singola istanza). Per questo, ogni oggetto dovrebbe essere completamente configurato prima di essere associato con un'etichetta atomica. Ciò ricondurrebbe allora l'approccio delle teorie parametriche ai problemi (inaccettabili, come discusso sopra) dell'opzione dell'istanza ground.

Esempio 2.8

Supponiamo che il simbolo d'equivalenza ≡ rappresenti la relazione di aliasing. Allora, i seguenti

```
r1 ≡ resistor(12, 4, 3).

r2 ≡ resistor(12, 2, 6).
```

sono assiomi (dove si può leggere ≡ come le relazioni `<=` o `<` di L&O [McC92]) che dichiarano `r1` e `r2` come istanza di `resistor`, così che questi termini possono essere usati ovunque in un programma per indicare le

corrispondenti istanze di `resistor`. Quindi, il goal

```
?- r2 <- equivalent(parallel(r1, r1)).
yes
```

ha successo. Peraltro, `r1` e `r2` devono essere completamente istanziati per essere usati, se si vuole che le dichiarazioni di aliasing siano assiomi a tutti gli effetti, e che ogni alias denoti un'unica istanza: per questo, `r1` e `r2` non possono essere utilizzati in computazioni volte alla loro configurazione. Goal come `g7-8`, `g10` non potrebbero pertanto essere mai usati qui.

2.3 Sommario

In conclusione, le teorie logiche, combinate in gerarchie, forniscono il miglior modo per esprimere concetti come classe, ereditarietà, e scambio di messaggi. Il problema fondamentale da risolvere allo scopo di costruire un modello di programmazione logica pienamente orientato agli oggetti è come rappresentare le istanze: cioè, come trattare lo stato.

Se un oggetto in programmazione logica è ciò che di esso sappiamo essere vero, la rappresentazione tramite assiomi è senz'altro la più naturale, consentendo la modellazione diretta dell'astrazione di oggetto. L'interpretazione a scambio di messaggi della richiesta di dimostrazione di un goal in un dato contesto di prova può essere esteso in linea di principio dai contesti alle istanze.

Inoltre, una rappresentazione assiomatica comune di metodi e attributi porta a un modello computazionale uniforme, che implica una serie di benefici. In primo luogo, l'uniformità dei meccanismi di binding consente un accesso uniforme all'informazione di stato e alla conoscenza procedurale da parte di tutti i metodi. Secondo, è possibile sfruttare un unico meccanismo per l'ereditarietà degli attributi e dei metodi. Terzo, i meccanismi di visibilità utilizzati per i metodi (come i filtri sui predicati) possono essere usati per gli

attributi, in modo da fornire uno strumento per l'information hiding. Quarto, la persistenza delle istanze può essere ottenuta in modo naturale, rafforzando ed estendendo così in senso object-oriented l'interpretazione di database dei programmi logici. Quinto, è quindi disponibile concettualmente (anche se non ancora chiarificato nelle sue implicazioni complessive) un modello per la modifica dello stato, senza la necessità di interferire con l'interpretazione dei simboli logici: associando nuovi assiomi al medesimo simbolo risulta possibile determinarne il cambiamento di stato.

In aggiunta, la rappresentazione di oggetti tramite coppie di identificatori ground (di livello oggetto) e teorie logiche (di metalivello) comporta diversi vantaggi. Prima di tutto, esiste una distinzione netta tra la denotazione di un oggetto e il suo stato, cosicché è possibile definire astrazioni di dato. La conoscenza relativa allo stato di un oggetto è incapsulata in un'unità d'informazione separata cui si può accedere direttamente per nome, e che può essere utilizzata come un blocco di conoscenza indipendente, eventualmente combinabile con altri blocchi. L'interpretazione di base di dati segna un percorso che conduce a un modello per la modifica dello stato: se ogni teoria logica rappresenta lo stato un oggetto *in un dato momento* dell'evoluzione temporale del dominio applicativo, allora una *successione di teorie logiche* (tutte associate al medesimo simbolo) può rappresentare l'evoluzione di un oggetto nel tempo. Infine, questo approccio consente di operare una chiara distinzione concettuale tra le operazioni di livello oggetto e quelle di metalivello.[9]

Perciò, nel prossimo capitolo discuteremo il framework contestuale esteso che ha funto base per il nostro modello logico orientato agli oggetti, mentre nei capitoli successivi tratteremo e risolveremo il problema della configurazione di stato in programmazione logica.

---

[9] Lo stesso non può dirsi di approcci come [McC92], in cui un'operazione di configurazione di un termine (di livello oggetto) può risultare in realtà la configurazione di una teoria quando si incontri un operatore di invio di messaggio che coinvolga come destinatario il termine stesso.

3 Programmazione logica contestuale in CSM

3.1 La programmazione logica contestuale

3.1.1 Programmazione logica e costruzione del software

Le necessità proprie della produzione industriale del software (modularità, componibilità, estendibilità, riusabilità) contrastano fortemente, come già accennato nei precedenti capitoli, con la visione del programma logico come

un componente software chiuso promossa dalla Assunzione di Mondo Chiuso (CWA).

In questo senso, un programma logico dovrebbe altresì consistere non tanto in un insieme di assiomi completo e immodificabile, quanto piuttosto in una collezione di componenti separati (*moduli*, *teorie*, o *unità*), che costituiscono porzioni di conoscenza relativa dominio applicativo, e aperti rispetto all'informazione contenuta in altri componenti.

A partire dal lavoro di Miller [Mil89], una quantità di diverse proposte per un'approccio modulare alla programmazione logica sono state proposte in letteratura: piuttosto che dilungarci in questa sede sull'argomento, ripetendo cose già note, preferiamo rimandare all'esauriente panoramica di [BLM94].

Ai nostri fini, invece, è sufficiente ricordare il significato del connettivo $\supset$ di *embedded implication*, introdotto da Miller. Se T è una collezione di clausole logiche universalmente quantificate, e G un goal, diciamo che l'*implicazione* $T \supset G$ può essere derivata dal programma P se G può essere dimostrato a partire dal programma P' ottenuto come unione di P e T.

Usando il classico formalismo dei *sequenti* (si veda anche la sottosezione 3.2.2) per le regole d'inferenza, possiamo quindi scrivere che

$$\frac{P \cup T \vdash G}{P \vdash T \supset G} \qquad (3.1)$$

Oltre a consentire la partizione dello spazio della conoscenze, l'implicazione permette quindi una prima, semplice forma di composizione di teorie logiche.

Più semplice l'approccio utilizzato in genere dai sistemi per la programmazione logica più diffusi, che non consentono la composizione diretta di teorie. In particolare, molti sistemi Prolog commerciali (tra cui SICStus Prolog [SICS] e Quintus Prolog) sfruttano il concetto di modulo come puro contenitore sintattico. L'apertura di questi componenti è ottenuta in primo luogo mediante meccanismi di import/export di predicati, e quindi con una forma di delegazione esplicita da un componente a un altro, realizzata non

per composizione di teorie, ma per *cambio di contesto*.

Se infatti diciamo M un modulo di un programma P, e reinterpretiamo il connettivo $\supset$ come cambio di contesto, allora il goal $M \supset G$ è vero rispetto a P se G può essere derivato dalle clausole di M. La regola seguente

$$\frac{M \vdash G}{P \vdash M \supset G} \tag{3.2}$$

esprime (per quanto rozzamente) questa idea.

Viceversa, la nozione di *teoria* logica è più fortemente connotata dal punto di vista semantico. Una teoria non è intesa come un semplice contenitore di clausole, bensì come la descrizione di un elemento del dominio applicativo. Ogni teoria, quindi, contiene (tutta e sola) l'informazione relativa a un'astrazione del dominio che ne costituisce quindi il corrispettivo semantico. In breve, la teoria di un oggetto è sintatticamente un modulo logico che ne descrive le proprietà tramite i suoi assiomi.

La constatazione d'incompletezza della conoscenza, insita nella nozione di programma come collezione di teorie logiche, conduce direttamente alla *composizione* delle teorie. La definizione incrementale (per specializzazione, generalizzazione, o quant'altro) degli elementi del dominio del discorso può essere ottenuta definendo meccanismi (gerarchici o *piatti*, statici o dinamici) per la combinazione di teorie logiche *aperte*.

3.1.2 Programmazione logica contestuale

La *programmazione logica contestuale* (che indicheremo spesso con *CtxLP* di qui in avanti) è stata introdotta da [MoPo89] come estensione alla programmazione logica classica che associa una peculiare nozione di ambito lessicale all'embedded implication [Mil89].

Denotiamo con *closure*(T) l'insieme degli atomi che possono essere derivati da una teoria logica T (o *unità*, secondo la terminologia tipica della CtxLP) sotto CWA, ossia

$$closure(T) = \{A \mid A \text{ atomico} \wedge T \vdash A\}$$

È palese che vale $closure(T) \subseteq B_T$ (ove B_T rappresenta la Base di Herbrand della teoria *T*).

Allora, il modello contestuale (CtxLP, d'ora in avanti) originale può essere formalizzato tramite la seguente regola di inferenza, che realizza la cosiddetta *estensione di contesto*:

$$\frac{U \lhd closure(T) \vdash G}{T \vdash U >>> G} \qquad (3.3)$$

in cui il simbolo $\lhd$ rappresenta l'*unione con overriding*: la teoria $T' \lhd T''$ contiene tutte le clausole di *T'* insieme alle sole clausole di *T"* relative a predicati non ridefiniti da *T'*. L'operatore >>> di *estensione di contesto* può quindi essere letto come una particolare versione del connettivo $\supset$.[10]

La chiusura (*closure*) di *T* assicura che il contributo di *T* alla dimostrazione di *G* non dipende dal componente *U*, che lo estende. L'overriding limita poi l'incidenza di *T* a quella parte di conoscenza che non appartiene a *U* (ossia, ai predicati che esso non definisce). Questo meccanismo di estensione consente al programmatore logico di gestire e controllare il ruolo di ogni modulo del programma, e gli fornisce uno strumento per la costruzione di sistemi gerarchici basati sulla conoscenza, in cui ogni strato può ereditare informazione dagli strati precedenti.

Adottando la visione procedurale delle clausole si può determinare la politica di binding associata all'estensione di contesto. Grazie alla chiusura, il binding dei predicati non locali a un dato strato software (un modulo) può essere stabilito all'atto dell'introduzione dello strato stesso, quindi *quasi-staticamente*. Questo produce una politica di binding *eager* ("impaziente") dei predicati non locali.

---

10 L'operatore (originariamente >>) è sintatticamente associativo a destra per consentire l'estensione incrementale di un sistema tramite valutazione da sinistra verso destra.

Esempio 3.1.

Supponiamo che il nostro programma sia costituito dalle seguenti 3 unità:

```
:- unit u1.          :- unit u2.          :- unit u3.
q(1).                p(X) :- q(X).        q(3).
```

La risoluzione del goal

```
?- u1 >>> u2 >>> u3 >>> p(X).
```

porta il sistema alla configurazione di programma della figura 3.1 prima della prova di `p(X)`. Nel programma risultante, la chiamata al predicato `q/1` in `u2` è legata permanentemente alla definizione contenuta in `u1` all'atto dell'estensione di `u1` con `u2`. La successiva estensione con `u3` nulla cambia nel comportamento del sistema, e la dimostrazione ha successo con `X=1`.

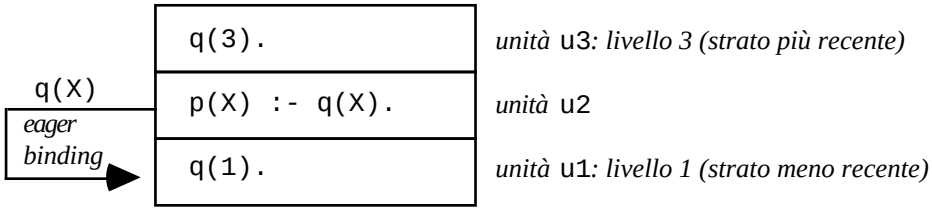

Figura 3.1.

Viceversa, la query

```
?- u3 >>> u2 >>> u1 >>> p(X).
```

produce la configurazione della figura 3.2 e la risposta calcolata `X=3`.

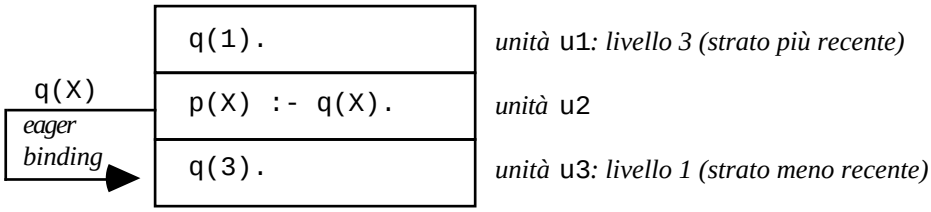

Figura 3.2.

La politica di binding eager non consente per altro di definire componenti software aperti. In particolare, il comportamento di uno strato software di

livello *h* non potrà mai dipendere dalle definizioni di strati più recenti (di livello *k*>*h*) tramite l'eager binding.

D'altra parte, il paradigma di programmazione a oggetti mostra quanto sia importante la capacità di costruire componenti aperti. In particolare, una classe può specializzare il comportamento di una superclasse sfruttando la politica di binding *late* (*ritardato*), o *lazy* ("pigro").

Per tale ragione, il modello CtxLP originale è stato esteso [MeNa92] combinando il binding dinamico con l'overriding:

$$\frac{U \lhd T \vdash G}{T \vdash U >>> G} \qquad (3.4)$$

In questo caso, l'insieme delle clausole da utilizzare nella prova di *G* è il risultato dell'unione con overriding delle clausole di *U* e *T*. Questa regola reintroduce la dipendenza bidirezionale tra le teorie *T* e *U*, e promuove una forma di binding lazy grazie alla quale ogni strato fa sempre uso delle definizioni più recenti che sono disponibili, anche se introdotte successivamente allo strato stesso tramite goal estensione.

Esempio 3.2.

Se il nostro programma si compone delle stesse tre unità dell'esempio 3.1, la risoluzione del goal

```
?- u1 >>> u2 >>> u3 >>> p(X).
```

porta ora il sistema all'architettura della figura 3.3, e alla risposta calcolata `X=3`. Qui, infatti, la chiamata al predicato `q/1` in `u2` è legata dinamicamente alla definizione (più recente) contenuta in `u3` all'atto della dimostrazione di `q(X)` in `u2`.

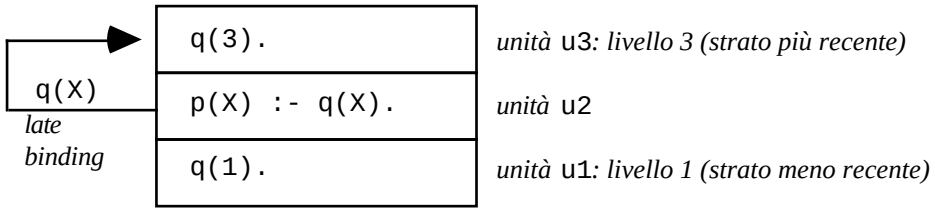

Figura 3.3.

Si può notare come, a differenza dell'esempio 3.1, l'estensione con `u3` cambi il comportamento del componente costituito dalle unità `u2` e `u1`. Evitando infatti di estendere con `u3`, ossia provando la query

```
?- u1 >>> u2 >>> p(X).
```

la risposta del sistema cambia, e la prova ha successo con `X=1` (si veda la figura 3.4).

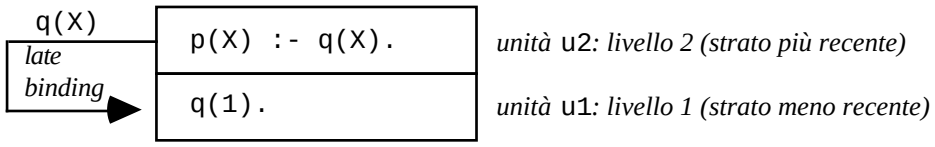

Figura 3.4.

Combinando quindi la composizione di teorie logiche per estensione con entrambe le politiche di binding risulta così possibile definire anche in programmazione logica sistemi software strutturati che sfruttino le tecniche di progetto tipiche della programmazione a oggetti.

3.2 CSM

3.2.1 CSM: il modello contestuale esteso

Il primo risultato di questa tesi è stata la definizione di un modello esteso per la programmazione logica contestuale, che accresce il paradigma logico di alcuni strumenti fondamentali per la strutturazione del software. Il modello è stato immediatamente istanziato come estensione al più classico e diffuso dei

linguaggi logici, ossia il Prolog. Di qui il fatto che le riflessioni su meccanismi, politiche e astrazioni, che seguono, saranno sempre accompagnate da corrispondenze sintattiche, anche per consentirne l'immediata comprensione con l'ausilio di brevi esempi.

Il primo criterio nello sviluppo del modello è quello di compatibilità con il modello classico della programmazione logica, così come realizzato ed esteso dal Prolog. Il primo punto del modello è quindi quello dell'esistenza di un default per il binding locale: ogni predicato definito in una teoria e non altrimenti denotato è risolto utilizzando la definizione contenuta nella teoria stessa. In questo modo, un programma Prolog qualunque conserva identico il suo comportamento in CSM, una volta preso e utilizzato come una singola unità logica.

Inoltre, da questo default consegue la necessità di denotare esplicitamente i predicati da risolvere non localmente: diciamo infatti che le unità di CSM sono teorie *open-denoted* ("aperte per denotazione"), e non semplicemente aperte, sulla falsariga di [BGLM92] e altri. Ciò corrisponde logicamente all'esigenza di un solido controllo sui "punti di apertura" dei moduli software, cosicché il comportamento di ogni componente dipenda dagli altri con cui viene composto in modo "disciplinato" dal programmatore del componente.

In aggiunta, definire due diversi generi di binding, locale e non locale, comporta la necessità di comprendere il tipo di binding tra le informazione di stato proprie della computazione logica contestuale.

In secondo luogo, come in [MeNa92], entrambe le politiche di binding non locale sono adottate in CSM. Con l'ausilio di appositi operatori, ogni chiamata di predicato può essere legata non localmente secondo una politica di binding eager o late, rispettivamente. In particolare, in CSM vengono introdotti gli operatori (prefissi) `xelf` e `#`, che consentono di vincolare un predicato ad una risoluzione del binding di tipo late, e l'operatore (prefisso) `xuper`, che permette invece di realizzare politiche di binding eager.

Per ottenere questa coesistenza, è necessario registrare complessivamente

nello stato della macchina logica estesa l'informazione relativa alla unità corrente dove si sta svolgendo la dimostrazione locale, e quella relativa alla struttura corrente del programma rispetto al quale la dimostrazione corrente deve essere svolta,. Tale struttura è a sua volta rappresentata in ogni momento da una sequenza di unità, che viene detta *contesto*.

Per questa ragione, vengono introdotti due elementi che fanno parte dello stato della computazione logica contestuale: il *contesto corrente*, costituito dall'elenco ordinato delle unità che compongono il programma logico corrente, e il *contesto di binding*, che è quella parte del contesto corrente che ha come unità "più recente" (*top unit*) l'unità corrente.

In aggiunta, definire due diversi generi di binding, locale e non locale, comporta la necessità di comprendere il tipo di binding tra le informazione di stato proprie della computazione logica contestuale. L'esempio che segue mostra l'evoluzione dello stato di una semplice computazione in CSM.

Esempio 3.3.

Adottando la tipica notazione a lista, secondo la quale un contesto costituito dalle unità u_n, u_{n-1}, …, u_2, u_1 può essere rappresentato tramite la lista $[u_n, u_{n-1}, \ldots, u_2, u_1]$, possiamo denotare lo stato della macchina logica contestuale. Supponiamo che il nostro programma sia costituito dalle seguenti unità:

```
:- unit ua.     :- unit ub.                  :- unit uc.
s(a).           p(X,Y) :- q(Y,X).            r(c).
                q(X,Y) :- xelf r(X),
                          xuper s(Y).
```

Allora, la computazione indotta dalla query seguente

```
?- ua >>> ub >>> uc >>> p(X,Y).
```

evolve secondo la sequenza della figura 3.5.

passo	*contesto corrente*	*contesto di binding*	*binding*	*risolvente*	*risposta calcolata*
1	`[]`	`[]`	non locale	`ua >>> ub >>> uc >>> p(X,Y)`	
2	`[ua]`	`[ua]`	non locale	`ub >>> uc >>> p(X,Y)`	
3	`[ub,ua]`	`[ub,ua]`	non locale	`uc >>> p(X,Y)`	
4	`[uc,ub,ua]`	`[uc,ub,ua]`	non locale	`p(X,Y)`	
5	`[uc,ub,ua]`	`[ub,ua]`	locale a `ub`	`p(X,Y)`	
6	`[uc,ub,ua]`	`[ub,ua]`	locale a `ub`	`q(Y1,X1)`	`X=X1, Y=Y1`
7	`[uc,ub,ua]`	`[ub,ua]`	locale a `ub`	**`xelf r(X2)`**, `xuper s(Y2)`	`Y1=X2, X1=Y2`
8	`[uc,ub,ua]`	`[uc,ub,ua]`	non locale	`r(X2), …`	
9	`[uc,ub,ua]`	`[uc,ub,ua]`	locale a `uc`	`r(X2), …`	
10	`[uc,ub,ua]`	`[uc,ub,ua]`	locale a `uc`	`□, …`	`X2=c`
11	`[uc,ub,ua]`	`[ub,ua]`	locale a `ub`	**`xuper s(Y2)`**	
12	`[uc,ub,ua]`	`[ua]`	non locale	`s(Y2)`	
13	`[uc,ub,ua]`	`[ua]`	locale a `ua`	`s(Y2)`	
14	`[uc,ub,ua]`	`[ua]`	locale a `ua`	`□`	`Y2=a`
				successo	`X=a, Y=c`

Figura 3.5. Computazione indotta dalla query contestuale `ua >>> ub >>> uc >>> p(X,Y)`.

Lo stato iniziale (passo 1) di una computazione contestuale consiste nei due contesti di stato (corrente e di binding) vuoti (`[]`, `[]`), con valutazione non locale (e risposta calcolata vuota, ovviamente).

La valutazione delle tre operazioni di estensione (passi 2, 3, 4) costruisce la teoria logica strutturata rispetto alla quale effettuare la prova del goal atomico `p(X,Y)`, ossia il contesto `[uc,ub,ua]`.

Il binding non locale di `p/2` è risolto con la definizione in `ub` (passo 5), che diviene l'unità corrente per il binding locale. Corrispondentemente, `[ub,ua]` diviene il contesto di binding.

La computazione procede poi localmente a `ub` come in un normale programma logico (passi 6, 7, 11), finché l'occorrere di operatori di binding contestuale (`xelf` e `xuper`) non forza la valutazione non locale di `r(X)` e `r(Y)`. Tali operatori non mutano il contesto corrente (che per entrambe gli atomi rimane `[uc,ub,ua]`), ma stabiliscono in quale sua sottoparte debba essere cercata la definizione per i goal atomici implicati.

In primo luogo, l'operatore `xelf` porta a considerare l'intero contesto corrente come fonte di definizioni per `r/1` (passo 8). In particolare, il binding viene risolto con la definizione trovata in `uc`, che diviene l'unità corrente (passo 9): di conseguenza, il contesto di binding è tutto `[uc,ub,ua]`. La relativa prova di `r(X)` prosegue poi secondo le normali regole del Prolog (passo 10).

In secondo luogo, l'operatore `xuper` riduce le definizione candidate al binding di `s/1` a quelle contenute nel contesto che si ottiene togliendo l'unità corrente in testa al contesto di binding: nel caso (passo 12), a quelle contenute nel contesto `[ua]`. In particolare, l'unità `ua` contiene effettivamente una definizione per `s/1`, e diviene quindi (passo 13) la nuova unità corrente (e il contesto di binding risultante è lo stesso `[ua]`). La computazione termina poi con successo secondo gli usuali canoni dei sistemi logici (passo 14), fornendo la risposta calcolata `{X=a,Y=c}`.

Uno degli elementi fondamentali dell'approccio contestuale è costituito dall'estensione di contesto. Secondo lo schema rappresentato dalle formule (3.3) e (3.4), essa consiste in due operazioni distinte:

i) estensione della teoria che rappresenta il programma corrente con un'unità che costituisce il primo argomento dell'operazione. La teoria risultante diviene quindi il nuovo contesto corrente per la computazione;

ii) prova del goal passato come secondo argomento nel nuovo contesto corrente. Ogni goal atomico viene quindi legato usando tutte le definizioni contenute nel contesto corrente, stabilendo corrispondentemente il contesto di binding.

D'altra parte, cosa sia la teoria *T* che rappresenta il programma corrente nelle formule (3.3) e (3.4) non è a priori noto nel modello contestuale esteso: così come fare riferimento al contesto corrente o al contesto di binding per il

binding non locale cambia il tipo di politica adottata nella composizione di moduli software aperti, la stessa alternativa sussiste per quanto riguarda l'operazione di estensione di contesto.

Come definito per esempio in [MeNa92], è possibile quindi definire due tipi di estensione di contesto: l'estensione *lineare* e quella *a cactus*. Facendo riferimento alle formule (3.3) e (3.4), ciò che cambia nei due casi è cosa si va a considerare come la teoria *T* che deve essere estesa. In un caso (estensione lineare) *T* è il contesto corrente; nel secondo (estensione a cactus) *T* è il contesto di binding.

Corrispondentemente, CSM definisce due operatori per l'estensione di contesto: `>>>` per l'estensione lineare, e `>:>` per l'estensione a cactus. Così, la dimostrazione del goal `u>>>g` procede stabilendo come contesto per la dimostrazione di `g` il contesto corrente esteso con l'unità `u`, mentre la dimostrazione del goal `u>:>g` determina come contesto corrente per la dimostrazione di `g` ciò che si ottiene estendendo il contesto di binding con l'unità `u`.

Adottando l'una o l'altra delle politiche di estensione, diversi sono i modelli architetturali dei sistemi risultanti. Con l'estensione lineare, il modello di riferimento è quello di una conoscenza che cresce in modo monotono con l'evolversi della computazione. Viceversa, l'estensione a cactus rimanda per esempio a modelli di ragionamento ipotetico, in cui si assumono, si sfruttano e quindi eventualmente si rilasciano le informazioni adottate come ipotesi secondo la linea di ragionamento corrente. Per maggiori dettagli sulle possibili architetture e sulle loro proprietà, si veda [BLM90].

A illustrare i meccanismi dell'estensione di contesto, si veda invece l'esempio seguente.

Esempio 3.4.

Supponiamo che il nostro programma sia costituito da 4 unità, tra cui le 3 seguenti,

```
:- unit ub.              :- unit uc.          :- unit ud.
q(X) :- xelf r(X).       r(c).                q(d).
```

e di dover risolvere il goal

```
?- ua >>> ub >>> p(X).
```

Se l'unità `ua` contiene un'unica clausola, il cui corpo è costituito da un'estensione di contesto lineare, come nel caso seguente,

```
:- unit ua.
p(X) :- uc >>> q(X).
```

allora la configurazione del programma generata dalla prova di `p(X)` è quello della figura (3.6).

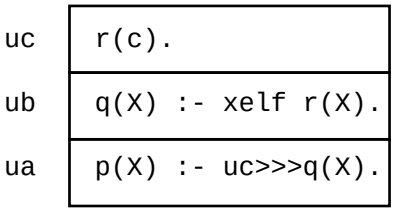

Figura 3.6.

passo	*contesto corrente*	*contesto di binding*	*binding*	*risolvente*	*risposta calcolata*
1	`[]`	`[]`	non locale	`ua >>> ub >>> p(X)`	
2	`[ua]`	`[ua]`	non locale	`ub >>> p(X)`	
3	`[ub,ua]`	`[ub,ua]`	non locale	`p(X)`	
4	`[ub,ua]`	`[ua]`	locale a `ua`	`p(X)`	
5	`[ub,ua]`	`[ua]`	locale a `ua`	`uc >>> q(X1)`	`X=X1`
6	`[uc,ub,ua]`	`[uc,ub,ua]`	non locale	`q(X1)`	
7	`[uc,ub,ua]`	`[ub,ua]`	locale a `ub`	`q(X1)`	
8	`[uc,ub,ua]`	`[ub,ua]`	locale a `ub`	`xelf r(X2)`	`X1=X2`
9	`[uc,ub,ua]`	`[uc,ub,ua]`	non locale	`r(X2)`	
10	`[uc,ub,ua]`	`[uc,ub,ua]`	locale a `uc`	`r(X2)`	
11	`[uc,ub,ua]`	`[uc,ub,ua]`	locale a `uc`	□	`X2=c`
				successo	`X=c`

Figura 3.7. Computazione indotta dalla query contestuale `ua >>> ub >>> p(X)`. (caso a)

Qui si vede come tutte le unità coinvolte nella computazione siano composte a

costituire il programma rispetto al quale viene effettuata la dimostrazione, secondo la politica di estensione lineare.

La computazione procede poi secondo i passi riportati dalla figura (3.7). Si noti in particolare l'effetto del passo 5 sullo stato della macchina, quando il goal estensione viene effettivamente valutato.

Se viceversa l'unità `ua` è definita nel modo seguente,

```
:- unit ua.
p(X) :- ud >:> q(X).
```

con il corpo dell'unica clausola costituito da un'estensione a cactus, allora la configurazione del programma generata dalla prova di `p(X)` è quello della figura (3.8).

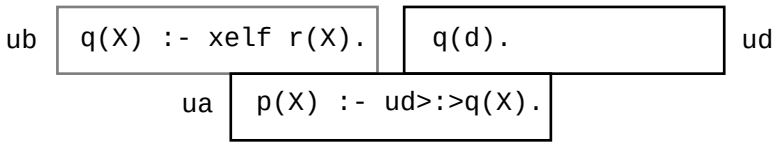

Figura 3.8.

passo	*contesto corrente*	*contesto di binding*	*binding*	*risolvente*	*risposta calcolata*
1	`[]`	`[]`	non locale	`ua >>> ub >>> p(X)`	
2	`[ua]`	`[ua]`	non locale	`ub >>> p(X)`	
3	`[ub,ua]`	`[ub,ua]`	non locale	`p(X)`	
4	`[ub,ua]`	`[ua]`	locale a `ua`	`p(X)`	
5	`[ub,ua]`	`[ua]`	locale a `ua`	`ud >:> q(X1)`	`X=X1`
6	`[ud,ua]`	`[ud,ua]`	non locale	`q(X1)`	
7	`[ud,ua]`	`[ud,ua]`	locale a `ud`	`q(X1)`	
8	`[ud,ua]`	`[ud,ua]`	locale a `ud`	□	`X1=d`
				successo	`X=d`

Figura 3.9. Computazione indotta dalla query contestuale `ua >>> ub >>> p(X)`. (caso b)

In questo caso, diversamente dal precedente, l'estensione a cactus fa sì che l'unità `ud` rimpiazzi l'unità `ub` per la dimostrazione di `q(X)`, invece di aggiungersi alle teorie precedentemente introdotte. La computazione procede quindi secondo i passi riportati dalla figura 3.9. Si noti in particolare come

evolve lo stato della macchina dopo il passo 5, in cui viene effettuata l'estensione a cactus.

La notazione a lista utilizzata per denotare lo stato della macchina contestuale è in realtà parte della sintassi CSM per la denotazione di un contesto: un contesto in CSM può essere direttamente rappresentato tramite la lista delle unità che lo compongono.

Un'estensione al modello contestuale introdotta da CSM sfrutta tale capacità espressiva del linguaggio fornendo la possibilità di specificare direttamente (e non per estensioni successive) il contesto di prova per una formula. Il valore di verità di una qualunque formula atomica può quindi essere determinato in relazione a un qualunque contesto, indipendentemente dal contesto di prova corrente al momento in cui la formula occorre durante una dimostrazione. Ciò consente di delegare la prova di qualunque formula a componenti software la cui definizione e le cui proprietà non siano in alcun modo correlate con lo stato di una computazione.[11]

L'operatore ← di *cambio di contesto* (*context switch*) consente di associare un contesto a una formula logica: la formula $C \leftarrow G$ è vera se G è vero rispetto al contesto C. L'esempio seguente mostra l'effetto dell'utilizzo di tale operatore in una computazione CSM.

Esempio 3.5.

Supponiamo che il nostro programma sia costituito da 4 unità, tra cui le 3 seguenti,

```
:- unit ua.                :-              :- unit uc.
p(X) :- [ub,uc]<-q(X).      unit ub.       q(X) :- xelf r(X).
                           r(b).
```

[11] Effetto analogo a questa estensione viene ottenuta tramite l'introduzione di una speciale unità *top* in [Lam90].

e di dover risolvere il goal

```
?- [ub, ua] <- p(X).
```

La computazione procede poi secondo i passi riportati dalla figura (3.10). Si noti in particolare l'effetto del passo 4 sullo stato della macchina, quando il viene effettuato un cambio di contesto, cosicché la formula `q(X)` può essere dimostrata in un contesto di prova `[ub,uc]` completamente indipendente dal contesto corrente `[ub,ua]`.

passo	*contesto corrente*	*contesto di binding*	*binding*	*risolvente*	*risposta calcolata*
1	`[]`	`[]`	non locale	`[ub,ua] <- p(X)`	
2	`[ub,ua]`	`[ub,ua]`	non locale	`p(X)`	
3	`[ub,ua]`	`[ua]`	locale a `ua`	`p(X)`	
4	`[ub,ua]`	`[ua]`	locale a `ua`	`[ub,uc] <- q(X1)`	`X=X1`
5	`[ub,uc]`	`[ub,uc]`	non locale	`q(X1)`	
6	`[ub,uc]`	`[uc]`	locale a `uc`	`q(X1)`	
7	`[ub,uc]`	`[uc]`	locale a `uc`	`xelf r(X2)`	`X1=X2`
8	`[ub,uc]`	`[ub,uc]`	non locale	`r(X2)`	
9	`[ub,uc]`	`[ub,uc]`	locale a `ub`	`r(X2)`	
10	`[ub,uc]`	`[ub,uc]`	locale a `ub`	□	`X2=c`
				successo	`X=c`

Figura 3.10. Computazione indotta dalla query `[ub,ua] <- p(X)`.

La necessità di fornire meccanismi per la protezione dell'informazione ha poi portato alla definizione di regole di visibilità per gli assiomi di una teoria. Risulta così possibile in CSM costruire componenti software dotati di un'interfaccia verso l'esterno che limita l'accesso dall'esterno alle sole risorse che il componente vuole rendere disponibili.

L'information hiding in CSM si fonda sulla possibilità di definire diverse categorie di visibilità per i predicati. La granularità dei predicati è stata preferita alle altre possibili perché forniva il miglior compromesso tra una soluzione troppo "fine" (visibilità definita relativamente alle singole clausole) e una troppo "lasca" (visibilità definita rispetto a "sottoteorie", tipo "teoria di

interfaccia"). Ovviamente, in un ambiente a teorie multiple la granularità dei predicati è implicitamente più fine di quella di una teoria. Ogni proprietà di un predicato è quindi relativa a una singola teoria logica: lo stesso predicato `p` può quindi godere della proprietà *P1* nella teoria `t1`, e della proprietà *P2* nella teoria `t2`.

Le categorie di visibilità di CSM sono tre, e comprendono i predicati *visibili*, *protetti*, e *nascosti*. L'appartenenza di un predicato a una data categoria viene determinata da dichiarazioni della seguente forma poste all'interno della dichiarazione di una teoria, e che (coerentemente a quanto detto sopra) hanno ambito limitato alla stessa teoria in cui occorrono:

```
:- visible <predicati>.
:- protected <predicati>.
:- hidden <predicati>.
```

La prima categoria, quella dei predicati visibili, è quella implicitamente associata da CSM a ogni predicato in assenza di dichiarazioni. Gli esempi visti sin qui, non presentando dichiarazioni di sorta, comprendevano dunque solo predicati visibili. Se un predicato è visibile in una certa unità, la definizione corrispondente risulta disponibile per qualunque binding, locale o meno.

Le definizioni relative ai predicati nascosti, invece, non sono disponibili all'esterno di una unità. Quindi, ogni forma di binding non locale non può sfruttare tali definizioni: i predicati nascosti di una teoria possono essere utilizzati solo per il binding locale.

Più complessa la natura dei predicati protetti. Essi nascono sulla falsariga dei predicati *protected* del C++, ossia per definire proprietà che siano visibili all'interno di una gerarchia ma non all'esterno. Ciò corrisponde in CSM a introdurre una duplice politica di binding contestuale (non locale), che diremo rispettivamente *interna* ed *esterna*. In pratica, un'operazione di cambio di contesto implica un binding contestuale esterno (ci si rivolge a una gerarchie di teorie "dall'esterno"), mentre gli operatori di binding eager e lazy, nonché

le estensioni di contesto, implicano un binding contestuale *interno*.

CSM definisce, infine, tutta una serie di regole di default per determinare le proprietà di quei predicati non coinvolti esplicitamente in una dichiarazione di visibilità, su cui non ci soffermiamo. Per maggiori dettagli, si consulti il manuale dell'utente di CSM [CSM].

La composizione di unità nella programmazione contestuale, come abbiamo visto (formule 3.3 e 3.4), si fonda su di una politica di overriding dei predicati: estendere un contesto contenente una definizione per il predicato `p` con un'unità `u` che ne contiene una propria a sua volta conduce alla ridefinizione del predicato `p` secondo `u`.

Tuttavia, CSM consente di adottare per il binding contestuale dei predicati una politica di *estensione*, alternativa all'overriding. In CSM è quindi possibile dichiarare un predicato definito in un'unità come predicato *esteso*, tramite la dichiarazione

```
:- extended <predicati>.
```

Il binding contestuale di un predicato esteso può quindi essere risolto nell'unità in cui esso è stato definito esteso, ma anche altrove, sfruttando il non determinismo tipico dei linguaggi logici. Per esempio, se `p` è esteso nell'unità `usucc`, e il contesto corrente, che già contiene una definizione per `p` in `uprec`, viene esteso con `usucc`, il binding non locale di `p` rispetto al contesto così esteso potrà utilizzare nondeterministicamente sia la definizione di `usucc`, sia quella di `uprec`.

D'altra parte, per rinforzare l'idea di località, e di controllo del programmatore sull'apertura delle teorie e sulla loro composizione, la proprietà di estensione di un predicato non ha effetto sul binding locale.

Infine, è possibile modificare la politica di default di CSM tramite apposite dichiarazioni (cfr. [CSM]) da overriding a estensione. Per tale caso, CSM mette a disposizione una dichiarazione del tipo

```
:- overriding <predicati>.
```

duale della dichiarazione di estensione, e dall'ovvio significato.

3.2.2 CSM: la semantica operazionale

Al fine di adiuvare la comprensione precisa dei meccanismi e delle astrazioni introdotte sin qui, e illustrate solo tramite esempi e spiegazioni informali, introduciamo la specifica della semantica operazionale della programmazione logica contestuale in CSM nella forma canonica delle *regole di inferenza*, espresse in forma di sequenti.

Un sequente denota la derivabilità di una formula da una teoria logica: la seguente notazione

$$P \vdash_{\vartheta} G$$

rappresenta la derivabilità della formula G dalla teoria P con sostituzione ϑ (cfr. [CLM91]): data la collezione P di clausole di Horn, è possibile trovare una procedura di prova della formula G, sostituendo le variabili di G secondo la risposta calcolata ϑ.

In CSM, l'insieme delle clausole rispetto a cui una formula deve essere derivata cambia durante la computazione, e può essere denotato con la coppia dei contesti di stato: contesto corrente e contesto di binding. Inoltre, tale insieme varia anche a seconda del tipo di binding che di volta in volta deve essere applicato.

Dunque, un sequente di CSM avrà la forma seguente:

$$C \; {}_{C'}\,{}_{B}\vdash_{\vartheta} G$$

Qui, C e C' rappresentano rispettivamente il contesto corrente e il contesto di binding, mentre B rappresenta la modalità di binding. In particolare, h rappresenta il binding locale (quando cioè sono utilizzabili tutte le definizioni in una teoria, anche quelle dei predicati *hidden*), p il binding contestuale interno (sono accessibili le clausole dei predicati *protected* di un contesto, oltre a quelli visibili), e v il binding contestuale esterno (solo predicati

visible); la generica lettera *B* indica la modalità di binding generica.

È possibile derivare una formula $G\vartheta$ da un programma CSM se esiste una *prova* per $\varnothing_{ctx}\ \varnothing_{ctx}\ {}_{v}\vdash_{\vartheta} G$, dove $\varnothing_{ctx}$ rappresenta il contesto vuoto. Una prova per $C\ C'\ {}_{B}\vdash_{\vartheta} G$ è un albero di sequenti, la cui radice è $C\ C'\ {}_{B}\vdash_{\vartheta} G$, le foglie sono vuote, ogni nodo è legato ai suoi figli dalla relazione antecedente/conseguente secondo una delle regole di inferenza seguenti.

La regola (1) stabilisce la verità della formula *true* in ogni contesto. Qui, ε rappresenta la sostituzione vuota.

$$\frac{}{C\ C'\ {}_{B}\vdash_{\varepsilon} \texttt{true}} \tag{1}$$

La regola (2), invece, mostra come una congiunzione di formule possa essere dimostrata provando separatamente ciascuna formula.

$$\frac{C\ C'\ {}_{B}\vdash_{\sigma} G_i \qquad C\ C'\ {}_{B}\vdash_{\vartheta} \left(G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_k\right)\sigma}{C\ C'\ {}_{B}\vdash_{\sigma\vartheta} G_1, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_k} \tag{2}$$

Le regole seguenti stabiliscono invece i meccanismi di binding locale e non locale (contestuale). In primo luogo, il binding locale è analogo ai linguaggi logici classici, tipo Prolog, come mostra la regola (3):

$$\frac{C\ T \lhd C'\ {}_{h}\vdash_{\vartheta} G\sigma}{C\ T \lhd C'\ {}_{h}\vdash_{\sigma\vartheta} \texttt{p}(\tilde{\texttt{t}})} \qquad A_i \text{ :- } G \in T, \sigma = mgu\left(A_i, \texttt{p}(\tilde{\texttt{t}})\right) \tag{3}$$

Qui, $\tilde{\texttt{t}}$ rappresenta una *tupla* di termini, costituenti gli argomenti del predicato *p*. *T* è l'unità corrente, ossia la top unit del contesto di binding $T \lhd C'$. Questa particolare notazione significa semplicemente che il contesto di binding (ovviamente un sottocontesto del contesto corrente *C*) può essere letto come se fosse ottenuto per estensione del contesto *C'* con l'unità *T*. La funzione *mgu* è la classica funzione che restituisce l'unificatore più generale (*most general unifier*) tra due atomi, nel caso tra l'atomo da dimostrare localmente e

la testa di una delle clausole dell'unità corrente *T*.

Come si vede dalle regole successive, tutte le diverse forme di binding si riconducono alla forma fondamentale del binding locale. La regola (4) mostra come le invocazioni non locali di una procedura logica vengono ricondotte alla risoluzione locale.

$$\frac{C_{\ T \lhd C'' \ {}_{h}\vdash_{\vartheta}\ \mathtt{p}(\tilde{\mathtt{t}})}}{C_{\ C' \ {}_{B}\vdash_{\vartheta}\ \mathtt{p}(\tilde{\mathtt{t}})}} \qquad (T \lhd C'', \mathtt{p}) \in \Pi^{B}_{C'}, (B{=}v) \vee (B{=}p) \quad (4)$$

Entrambe le forme di binding contestuale, interno ($B=p$) ed esterno ($B=v$), sono fondate come si vede sull'insieme delle definizioni visibili e/o protette associate a ciascun contesto. A ogni contesto *C* sono quindi associati due *insiemi di binding*, costituiti da coppie contesto-predicato: l'*insieme di binding contestuale esterno*, denotato da Π^{v}_{C}, e l'*insieme di binding contestuale interno*, denotato da Π^{p}_{C}.

In particolare, la definizione di tale insieme si basa sulle proprietà dei predicati associati a un'unità. Data una unità *T*, diciamo T_d l'insieme dei predicati definiti localmente in *T*. Diciamo inoltre T_v, T_p, e T_h, rispettivamente, gli insiemi dei predicati visibili, protetti e nascosti di *T*. Diciamo infine T_o e T_e, rispettivamente, gli insiemi dei predicati overriding ed estesi di *T*. Per tali insiemi valgono ovviamente le seguenti proprietà:

$T_v \cap T_p = T_v \cap T_h = T_h \cap T_p = \varnothing$
$T_v \cup T_p \cup T_h = T_d$
$T_o \cap T_e = \varnothing$
$T_o \cup T_e = T_d$

Definiamo inoltre una serie di insiemi che risulteranno utili nel seguito:

$T_{pv} ::= T_p \cup T_v$
$T_{ov} ::= T_o \cap T_v$
$T_{opv} ::= T_o \cap T_{pv}$

Possiamo ora definire gli insiemi di binding in modo ricorsivo, partendo da due considerazioni: *i*) gli insiemi di binding associati al contesto vuoto sono a loro volta vuoti; *ii*) ogni altro contesto può essere pensato come la composizione di un'unità e di un contesto. Dato quindi un contesto C e una unità T, gli insiemi di binding associati al contesto $T \triangleleft C$ sono definiti come segue:

$$\Pi^{v}_{T \triangleleft C} ::= \left\{ \left(T \triangleleft C, \mathtt{p} \right) \;\middle|\; \mathtt{p} \in T_{v} \right\} \cup \left\{ \left(C', \mathtt{p} \right) \;\middle|\; \left(C', \mathtt{p} \right) \in \Pi^{v}_{C} \wedge \mathtt{p} \notin T_{ov} \right\}$$

$$\Pi^{p}_{T \triangleleft C} ::= \left\{ \left(T \triangleleft C, \mathtt{p} \right) \;\middle|\; \mathtt{p} \in T_{pv} \right\} \cup \left\{ \left(C', \mathtt{p} \right) \;\middle|\; \left(C', \mathtt{p} \right) \in \Pi^{p}_{C} \wedge \mathtt{p} \notin T_{opv} \right\}$$

In pratica, ogni unità aggiunge *tutte* le sue definizioni (protette e/o visibili) a quelle definizioni che, già presenti nel contesto con cui è composta, non collidano con i suoi predicati overriding. Gli insiemi risultanti vengono poi utilizzati nella regola (4) per descrivere i meccanismi di binding non locale: in particolare, si può vedere come sono sempre gli insiemi relativi al contesto di binding (da cui il nome) che vengono utilizzati per la risoluzione del binding contestuale.

La semantica operazionale degli operatori di binding contestuale può essere espressa secondo le regole (5) e (6).

$$\frac{C \; {}_{C}\!\vdash_{\vartheta}^{B'} \; G}{C \; {}_{C'}\!\vdash_{\vartheta}^{B} \; \mathtt{xelf}\; G} \; \left(B = B' = v \right) \vee \left(\left(B \neq v \right) \wedge \left(B' = p \right) \right) \qquad (5)$$

$$\frac{C \; {}_{C'}\!\vdash_{\vartheta}^{B'} \; G}{C \; {}_{T \triangleleft C'}\!\vdash_{\vartheta}^{B} \; \mathtt{xuper}\; G} \; \left(B = B' = v \right) \vee \left(\left(B \neq v \right) \wedge \left(B' = p \right) \right) \qquad (6)$$

La regola (7) descrive l'operazione di cambio di contesto:

$$\frac{C \; {}_{C}\!\vdash_{\vartheta}^{v} \; G}{C' \; {}_{C''}\!\vdash_{\vartheta}^{B} \; C \leftarrow G} \qquad (7)$$

Invece, l'estensione di contesto può essere caratterizzata come segue:

$$\frac{T \lhd C \;\; {}_{T \lhd C}\;{}_{B'}\!\vdash_{\vartheta}\; G}{C \;\; {}_{C'}\;{}_{B}\!\vdash_{\vartheta}\; T >>> G} \;\; (B = B' = v) \vee \big((B \neq v) \wedge (B' = p)\big) \quad (8)$$

$$\frac{T \lhd C' \;\; {}_{T \lhd C'}\;{}_{B'}\!\vdash_{\vartheta}\; G}{C \;\; {}_{C'}\;{}_{B}\!\vdash_{\vartheta}\; T >:> G} \;\; (B = B' = v) \vee \big((B \neq v) \wedge (B' = p)\big) \quad (9)$$

Sia gli operatori di binding sia gli operatori d'estensione agiscono modificando il contesto di prova per la formula a cui sono applicati. In particolare, mentre `xuper` e `xelf` si limitano a cambiare il contesto di binding (e di conseguenza l'unità corrente), gli operatori di cambio di contesto e di estensione portano direttamente alla modifica del contesto corrente, a cui il contesto di binding viene poi uguagliato.

3.2.3 CSM: la semantica dichiarativa

La semantica dichiarativa di CSM può essere espressa seguendo schemi differenti. L'idea di base della semantica dichiarativa è di arrivare a denotare un programma nei termini dell'insieme delle verità atomiche che derivano da esso.

Si adotterà qui lo schema basato sui *modelli di Herbrand ammissibili*, ispirato a quello utilizzato in [BLM91, BLM92a, BLM92b]. Esso consiste nel caratterizzare ogni entità di un programma contestuale (prima le unità, poi i contesti) con modelli la cui validità è condizionata da ipotesi, e ridurre gradatamente l'incidenza delle ipotesi sino ad arrivare a un modello unico per tutto il programma, non più condizionato da alcunché.

La complessità dell'esposizione consiglia qui di evitare l'introduzione dei predicati protetti, considerando solamente la distinzione tra i predicati visibili e quelli nascosti. Ciò non cambia nulla dal punto di vista concettuale né da quello sostanziale (il passaggio a uno schema che tenga conto anche dei predicati protetti consiste essenzialmente in un "raddoppio" di tutte le strutture da definire), ma semplifica di molto l'esposizione dei diversi passi.

La caratterizzazione semantica procederà secondo i seguenti passi:

i) denotazione di ogni unità tramite un insieme di modelli di Herbrand ammissibili, ciascuno condizionato da una serie di ipotesi relative alle altre teorie logiche;

ii) denotazione di ogni contesto tramite un insieme di modelli di Herbrand (prima modelli ammissibili, poi modelli ammissibili gerarchici), ciascuno condizionato da una serie di ipotesi sull'interazione con altri contesti;

iii) denotazione di un programma come insieme di contesti ciascuno denotato da un modello di Herbrand.

In CSM, un programma P è costituito da insieme finito di unità. Se T è un'unità di P, scriveremo $T \in P$. Data una unità T, diciamo poi *ground*(T) la teoria logica formata da tutte le clausole che sono istanze ground delle clausole di T. Sia H_P l'Universo di Herbrand costruito a partire dalle costanti e dai simboli di funzione di P (H_P è quindi unico per ogni programma CSM, e comprende anche i nomi delle unità), e B_T la Base di Herbrand costruito applicando a H_P i simboli di predicato di T (B_T è quindi proprio di ogni unità del programma, e cambia da unità a unità).

Ogni unità interagisce con le altre per mezzo dei meta-operatori contestuali, che ne denotano i "punti di apertura". In particolare, risulta utile caratterizzare tali punti, che diciamo *goal aperti*, per mezzo di appositi insiemi. Ogni unità CSM può contenere 5 tipi di goal aperti: *C*`<-`*G*, *U*`>>>`*G*, *U*`>:>`*G*, `xelf` *G*, `xuper` *G*. Data un'unità T, possiamo quindi definire *Open*(T) come l'insieme dei goal aperti che compaiono in *ground*(T). Definiamo quindi *OutOpen*(T), *UpOpen*(T), e *DownOpen*(T) come partizioni di tale insieme, per cui i goal aperti di tipo *C*`<-`*G* appartengono a *OutOpen*(T) (e vengono detti goal *out-open*), quelli `xelf` *G* e *U*`>>>`*G* a *UpOpen*(T) (e vengono detti goal *up-open*), mentre quelli `xuper` *G* e *U*`>:>`*G* fanno parte di *DownOpen*(T) (e vengono detti goal *down-open*). È quindi ovvio come risulti

$$OutOpen(T) \cap UpOpen(T) = OutOpen(T) \cap DownOpen(T) = \\ = UpOpen(T) \cap DownOpen(T) = \varnothing$$

$$OutOpen(T) \cup UpOpen(T) \cup DownOpen(T) = Open(T)$$

L'insieme dei goal aperti di un'unità può anche essere letto come l'insieme delle possibili *ipotesi* su cui si può fondare la dimostrazione di formule nella teoria *T*. Per questo, nel seguito parleremo indifferentemente di goal aperti o di ipotesi, facendo riferimento agli elementi di $Open(T)$.

Possiamo quindi procedere con il primo passo, che consiste nella denotazione di ogni unità con un insieme di modelli di Herbrand.

Definizione 3.1. *Modello ammissibile di un'unità***.**

Sia *P* un programma CSM, e *T* un'unità di *P* ($T \in P$). Sia *H* un insieme di ipotesi per *T* ($H \subseteq Open(T)$). Sia quindi $ground(T)/H$ la teoria che si ottiene cancellando dalle clausole di $ground(T)$ tutte le ipotesi che compaiono in *H*, e cancellando quindi tutte le clausole che al termine di questa operazione contengono ancora goal aperti (che quindi non appartengono ad *H*). Avendo eliminato tutti i goal aperti, e quindi tutti gli operatori contestuali, la teoria risultante è quindi un'usuale teoria logica del prim'ordine, a cui possono essere applicate tutte le nozioni classiche.

Definiamo allora *modello ammissibile di Herbrand* dell'unità *T* secondo le ipotesi *H* il modello minimo di Herbrand di $ground(T)/H$, che denoteremo con $M_T(H)$.

È chiaro che vale $M_T(H) \subseteq B_T$. Diciamo inoltre $AHM(T)$ l'insieme dei modelli ammissibili di *T*, ossia

$$AHM(T) ::= \{M_T(H) \mid H \subseteq Open(T)\}$$

Il secondo passo consiste quindi nella composizione dei modelli delle singole teorie per ottenere una prima denotazione per i contesti. Componendo i

modelli ammissibili delle unità di un contesto, otteniamo in prima istanza ancora una denotazione in termini di modelli ammissibili, ossia condizionati a ipotesi. La composizione ci consente di eliminare in prima battuta i goal down-open, cancellando le ipotesi di tipo `xuper` *G* e trasformando quelle di tipo *U*>:>*G* in goal out-open.

Come tutte le definizioni concernenti i contesti, la strada più semplice è quella della definizione ricorsiva. Il punto di partenza è, come solito, il contesto vuoto, per il quale diamo per definizione la base di Herbrand, l'insieme delle ipotesi, e l'insieme dei modelli ammissibili, di seguito.

$B_{\emptyset_{ctx}} ::= \emptyset$
$Open(\emptyset_{ctx}) ::= \emptyset$
$AHM(\emptyset_{ctx}) ::= \emptyset$

Su questa base possiamo quindi definire gli stessi insiemi per un qualunque contesto: dato un contesto *C*, definiremo la sua base di Herbrand B_C, il suo insieme di ipotesi ammissibili *Open*(*C*), l'insieme *AHM*(*C*) dei suoi modelli di Herbrand ammissibili (ciascuno dei quali sarà denotato come $M_C(H)$ per le rispettive ipotesi *H*) in modo ricorsivo, partendo dalle definizioni analoghe per le unità e il contesto vuoto.

Definizione 3.2. *Modello ammissibile di un contesto***.**

Sia data $T \in P$, e il contesto *C*. Definiamo la Base di Herbrand e l'insieme delle ipotesi di $T \lhd C$ tali che

$B_{T \lhd C} ::= B_T \cup B_C$

$Open(T \lhd C) ::= Open(C) \cup OutOpen(T) \cup UpOpen(T) \cup DownToOut(DownOpen(T), C)$

dove *DownToOut* è una funzione che restituisce un insieme di goal out-open ottenuti trasformando tutti i goal *U*>:>*G* che appartengono a *DownOpen*(*T*) in goal out-open $U \lhd T \lhd C <\!\!- G$. Si noti come $Open(T \lhd C)$ non contiene goal down-open per costruzione.

Sia ora H un set di ipotesi per $T \lhd C$ ($H \subseteq Open(T \lhd C)$), e diciamo $H_C ::= H \cap Open(C)$, e $H_T ::= H \cap Open(T)$. Costruiamo quindi il corrispondente set di ipotesi $H_{T/C}$ (che può essere letto come "H di T dato C") che possono essere assunte per T dato H, e data la composizione di T con C: in particolare, possono essere valutati i goal di $DownOpen(T)$, che non fanno parte di $Open(T \lhd C)$. Così, fanno parte di $H_{T/C} \subseteq DownOpen(T)$ i goal di tipo `xuper` G che sono veri in C dato H (ossia, appartengono a $M_C(H_C)$), nonché quelli di tipo $U >: > G$ che, potendo qui essere considerati alla stessa stregua di goal out-open $U \lhd T \lhd C <\text{-} G$, sono accettabili se e solo se rientrano, con la loro forma out-open, tra le ipotesi H. Formalmente, definiamo $H_{T/C}$ come

$$H_{T/C} ::= \{\texttt{xuper}\ G \in Open(T) \mid G \in M_C(H_C)\} \cup$$
$$\{U >: > G \in Open(T) \mid U \lhd T \lhd C <\text{-} G \in H\}$$

Dato H, allora, ciò che risulta vero in T è $M_{T/C}(H) ::= M_T(H_T \cup H_{T/C})$, mentre il contributo di C è dato da $M_C(H_C)$. Di conseguenza, il modello ammissibile di $T \lhd C$ secondo le ipotesi H, ottenuto per composizione dei modelli ammissibili di T e C, è

$$M_{T \lhd C}(H) ::= M_{T/C}(H) \oplus_{\pi(T)} M_C(H_C)$$

L'operatore speciale di composizione $\oplus_{\pi(T)}$, corrisponde all'unione pura e semplice solo in caso di predicati tutti visibili e tutti estesi. Viceversa, l'operatore dispiega il suo effetto a seconda delle proprietà dei predicati della teoria T, per cui rappresenta in effetti una classe di operatori, così come suggerito dalla notazione. Esso infatti esclude dall'unione tutti gli atomi del primo argomento (nel caso, $M_{T/C}(H)$) il cui simbolo di predicato non appartiene a T_v (ossia, non è visibile in T), e tutti gli atomi del secondo argomento (nel caso, $M_C(H_C)$) il cui simbolo di predicato appartiene a T_{ov} (ossia, i predicati di C che devono essere sovrascritti dai predicati di T). L'operatore funge perciò sia da filtro per la visibilità dei predicati, sia da gestore delle politiche di estensione/overriding dei predicati.

I passi necessari a questo punto per arrivare a una denotazione unica per un programma sono ovvi, dato che per ogni contesto C siamo in grado ora di costruire l'insieme dei suoi modelli ammissibili $AHM(C)$ ciascuno dei quali dipende da ipotesi che sono solo up-open oppure out-open. Si tratta in primo luogo di *chiudere* ogni gerarchia di unità in modo da avere modelli dipendenti solo da ipotesi esterne, e quindi risolvere anche questi ultimi componendo tutti i modelli ottenuti.

Definizione 3.3. *Modello ammissibile gerarchico di un contesto***.**

Sia C un contesto di P, e sia $AHM(C)$ l'insieme dei suoi modelli ammissibili. Sia $HOpen(C)$ l'insieme delle ipotesi ammissibili, siffatto che

$$HOpen(C) ::= OutOpen(C) \cup UpToOut(UpOpen(C))$$

in cui *UpToOut* è una funzione che restituisce un insieme di goal out-open ottenuti trasformando tutti i goal $U>>>G$ che appartengono a $UpOpen(C)$ in goal out-open $U \triangleleft C<\text{-}G$. Si noti come $HOpen(C)$ contiene esclusivamente goal out-open per costruzione.

Si consideri $H \subseteq HOpen(C)$ come insieme di ipotesi. Se consideriamo C non più componibile con altre unità, possiamo valutare tutti le ipotesi del tipo $U>>>G$ che appartengono a $UpOpen(C)$ alla stessa stregua di goal out-open $U \triangleleft C<\text{-}G$. Definiamo perciò l'insieme H_{closed} a partire da H, in modo che sia

$$H_{closed} ::= \{U>>>G \in Open(C) \mid U \triangleleft C<\text{-}G \in H\}$$

Dato H, quindi, ciò che risulta vero per C è rappresentato da quei modelli ammissibili di C che si fondano su ipotesi di H, o H_{closed}, o su ipotesi di tipo `xelf` G in cui G sia già parte del modello. Definiamo quindi *modello gerarchico ammissibile* di C secondo H l'insieme $HM_C(H)$ tale che

$$HM_C(H) ::= \bigcup \{M_C(H') \mid$$
$$G' \in H' \Rightarrow ((G' \in H \cup H_{closed}) \vee ((G' = \texttt{xelf}\ G) \wedge (G \in M_C(H'))) \}$$

Diciamo anche $HAHM(C)$ l'insieme di tutti i modelli gerarchici ammissibili $HM_C(H)$ per il contesto C, ove ogni $H \subseteq HOpen(C)$ è un insieme di goal out-open.

Dati tutti i modelli gerarchici ammissibili, non resta che definire un operatore di composizione che, partendo da tali modelli, risolva tutti i goal out-open e fornisca un modello unico per il programma CSM.

È ovvio però che un modello per un programma CSM non potrà essere semplicemente un insieme di formule atomiche ground, ma dovrà invece essere costituito da coppie (*contesto*, *atomo ground*), poiché ogni formula in programmazione contestuale è vera o falsa solo rispetto a un ben determinato contesto di prova. Per questo, definiamo il concetto di *Base di Herbrand etichettata* per un programma CSM, come l'insieme delle coppie (*contesto*, *atomo ground*) appartenenti al programma stesso:

$$LB_P ::= \{(C, A) \mid (C \text{ è un contesto di } P) \wedge (A \in B_C)\}$$

Possiamo quindi definire un *operatore di composizione*, che effettui a ogni passo la chiusura di tutti quei goal out-open che possono essere assunti come veri, analogo a quello presentato nell'appendice B.

Definizione 3.4. ***Operatore di composizione*** ϕ_P**.**

L'operatore di conseguenza immediata ϕ_P è una funzione

$$\phi_P : 2^{LB_P} \rightarrow 2^{LB_P}$$

definita formalmente come segue:

$$\phi_P ::= \lambda L\ L \cup \{C{:}A \mid HM_C(H) \in HAHM(C) \wedge l(H) \subseteq L \wedge A \in HM_C(H)\}$$

in cui $l(H)$ è l'insieme di formule etichettate ottenute trasformando tutti i goal out-open C<-G in coppie (C, G).

È facile dimostrare che ϕ_P è *monotono* e *continuo*. La dimostrazione è semplice, e può essere condotta alla stessa stregua di quella relativa all'omonimo (e analogo) operatore ϕ_P nell'appendice B, per cui non si ritiene opportuno riportarla qui per esteso.

Per questo, ϕ_P ha un minimo punto fisso $lfp(\phi_P) = \phi_P{\uparrow}\omega$. Possiamo quindi definire la denotazione di un programma P di CSM nel modo seguente.

Definizione 3.5. *Modello per un programma CSM***.**

Sia P un programma CSM. Allora, il *modello etichettato* di P, che indichiamo con $||P||_{CSM}$, si ottiene applicando $\phi_P{\uparrow}\omega$ all'elemento minimo $\perp$ del reticolo completo 2^{LB_P}, che è ovviamente $\varnothing$. Definiamo quindi

$$||P||_{CSM} ::= \phi_P{\uparrow}\omega(\varnothing)$$

Per le proprietà dell'operatore, tale modello esiste ed è unico.

È chiaro che $||P||_{CSM} \subseteq LB_P$, per cui possiamo pensare di considerarlo come l'unione di un'infinità di modelli di Herbrand (uno per ogni contesto generabile a partire da P) ciascuno dei quali relativo a un singolo contesto.

Definiamo quindi *modello di Herbrand* di un contesto C di P il seguente insieme, ottenuto dal modello etichettato di P:

$$M_C ::= \{A \mid (C, A) \in ||P||_{CSM}\}$$

Ciò conclude ovviamente la caratterizzazione semantica dichiarativa di un programma CSM.

3.2.4 CSM: estensioni

La possibilità di definire gerarchie statiche di classi rappresenta uno dei fondamenti della programmazione a oggetti, in quanto consente di modellare direttamente in un programma le relazioni intercorrenti tra gli elementi del dominio applicativo.

Anche in CSM è possibile correlare tra loro classi, definendo relazioni tra unità logiche. CSM fornisce due diversi predicati (`isa` e `inherits`) che, utilizzati nella dichiarazione di testa di un'unità, consentono di associarle staticamente un'altra unità come sua *super unità*.

In particolare, la dichiarazione di un'unità, oltre alla forma vista sin ora,

```
:- unit T.
```

può quindi prendere le seguenti forme:

```
:- unit T isa S.
:- unit T inherits S.
```

Le due notazioni si distinguono soltanto per i loro effetti sul default da adottare per la politica di composizione delle unità (in particolare, `isa` e `inherits` stabiliscono rispettivamente estensione e overriding come politiche di default: si consulti [CSM] per maggiori dettagli).

Entrambe le notazioni, comunque, stabiliscono una (meta-)relazione *super* tra *T* ed *S*, che denotiamo con $T \propto S$. In questo modo, a ogni unità può essere associato un contesto, e precisamente quello risultante dalla chiusura transitiva della relazione super.

In particolare, se T_1 non ha super unità, il suo *contesto associato* è $T_1 \triangleleft \varnothing_{ctx}$, ossia il contesto formato dalla sola unità T_1. Se poi $T_k \propto T_{k\text{-}1}$ per qualunque k tale che $n \geq k > 1$, allora il contesto associato a T_k è $T_k \triangleleft T_{k\text{-}1} \triangleleft \ldots \triangleleft T_1 \triangleleft \varnothing_{ctx}$, ossia il contesto costituito dalla sequenza di unità T_k, $T_{k\text{-}1}$, …, T_1.

Ciò consente di adottare una notazione abbreviata per la denotazione di contesti, usando il nome di una unità in vece del suo contesto associato

laddove un nome di contesto sia atteso. Per esempio, il goal $T_k \leftarrow G$ equivale a un'operazione di cambio di contesto che forzi la prova di G nel contesto di prova $T_k \triangleleft T_{k-1} \triangleleft \ldots \triangleleft T_1 \triangleleft \varnothing_{ctx}$, associato a T_k.

Con questa semplice convenzione sintattica è quindi possibile inviare "messaggi" a componenti software elementari (le unità) che si comportano come se appartenessero una tassonomia predefinita, ereditando di fatto da altre entità (ossia, le unità appartenenti al contesto associato) comportamenti e proprietà, e specializzandole a sua volta.

Una convenzione sintattica ulteriore, anch'essa derivata dai linguaggi a oggetti più diffusi, è quella di considerare i predicati non definiti in una unità, ma usati nelle computazioni locali, come implicitamente preceduti dall'operatore `xuper`: i predicati *undefined*, quindi, non falliscono a priori, ma vengono risolti secondo una politica di binding eager.

Ciò consente di realizzare componenti che ereditano implicitamente da componenti meno specializzati comportamenti e proprietà che essi non ridefiniscono, così come accade tipicamente nei linguaggi a oggetti.

Infine, CSM è caratterizzato tutta una serie di predicati predefiniti che ne costituiscono l'ambiente di programmazione. I principali tra essi consentono l'ispezione delle proprietà statiche di un programma contestuale, nonché il controllo dello stato della macchina contestuale durante l'esecuzione di un programma. Tali (meta-)predicati dotano un programma CSM di capacità di autoanalisi sia statica sia dinamica. Per maggiori dettagli, si consulti ancora [CSM].

3.2.5 CSM: implementazione e semantica concreta

Il secondo risultato di questa tesi è stata la realizzazione del modello esteso per la programmazione logica contestuale (come definito nella sottosezione precedente) come estensione non intrusiva a un sistema industriale potente e diffuso per la programmazione logica.

Gli obbiettivi primari dell'implementazione, infatti, al di là della piena

realizzazione del modello presentato, possono essere riassunti come segue:

- non intrusività dell'estensione rispetto all'ambiente ospite: questo non deve perdere nulla (né in termini di espressività, né di strumenti disponibili, né di efficienza) a causa dell'estensione contestuale;
- efficienza dell'implementazione: tutte le operazioni svolte dal sistema esteso (e non delegate al sistema ospite) devono essere caratterizzate dalla massima velocità di esecuzione possibile;
- integrazione completa con l'ambiente ospite: l'estensione deve essere in grado di sfruttare al meglio gli strumenti di sviluppo forniti dall'ambiente ospite; inoltre, deve poter convivere con tutte le altre estensioni "*polite*" del sistema stesso;
- industrializzazione dell'ambiente di programmazione contestuale: il sistema deve essere dotato di ausili per lo sviluppo del software che supportino al meglio le estensioni introdotte; inoltre, deve essere dotato di un esauriente manuale, con esempi di programmazione, e di un aiuto in linea.

La scelta del supporto è quindi caduta sull'ambiente di programmazione logica SICStus Prolog [SICS]. Tale scelta è stata determinata da una serie di motivazioni: l'efficienza del codice prodotto, la possibilità di fare convivere codice compilato, codice interpretato e codice scritto in linguaggi diversi, la disponibilità di un'ampia gamma di librerie, il supporto, la disponibilità del codice sorgente, nonché l'amplissima diffusione sia a livello accademico sia a livello industriale.

Dopo un'attenta analisi delle tecniche alternative di implementazione note (meta-interpretazione, traduzione, estensione della WAM: per una discussione dettagliata, si vedano [LMNO91, DLMNO93]), si è individuata una strada originale che ha consentito di raggiungere pressoché tutti gli obbiettivi proposti.

Tale metodica, la cosiddetta *estensione basata sulla rappresentazione dei programmi*, si fonda sulla trasformazione statica del sorgente CSM in una serie di strutture di programma, alcune native di SICStus[12], altre ausiliarie. Il sistema CSM risultante delega poi tutto il peso della computazione al supporto SICStus, che sfrutta le strutture native appositamente create, intervenendo solo quando necessario. In particolare, il superstrato CSM interviene quando sono coinvolti operatori contestuali, che utilizzano ed eventualmente modificano le strutture ausiliarie, per poi ricondurre le invocazioni non locali a computazioni locali, che sono demandati di nuovo a SICStus. Per maggiori dettagli, si vedano [DNO92] e [Bolc94].

Ovviamente, la scelta di un supporto basato su Prolog ha portato con sé tutta una serie di conseguenze: in primo luogo, ha condotto all'adozione di una strategia di ricerca sull'albero di derivazione tipica di Prolog, ossia quella *in profondità* (*depth-first*). In qualche modo, quindi, la semantica operazionale descritta nella sottosezione precedente è stata specializzata per adattarsi alle modalità di computazione tipiche del linguaggio ospite, il Prolog.

Ciò pone in risalto il problema principale della *semantica concreta* di CSM, intesa come il comportamento effettivo della macchina CSM realmente implementata, che la genericità della semantica operazionale non consentiva di apprezzare: il problema della denotazione incompleta dei contesti di prova. Cosa succede in CSM, per esempio, se il nome di un'unità non è specificato al momento in cui essa deve estendere il contesto corrente, ossia se il primo argomento di un'estensione di contesto è una variabile non istanziata al momento dell'esecuzione?

Non avendo sin qui definito strumenti concettali per trattare questa situazione, ogni soluzione è ugualmente accettabile e insoddisfacente: sospendere la risoluzione del goal in attesa dell'opportuna istanziazione, provare tutte le possibili istanziazioni in maniera esaustiva, produrre

---

[12] Originariamente, moduli SICStus, da cui il nome CSM, come *C*ontexts as *S*ICStus *M*odules.

fallimento. Questa ultima soluzione, che sostanzialmente vieta l'uso di variabili non istanziate (in fase d'esecuzione) per denotare unità e contesti, è quella adottata dalla prima versione di CSM, se non altro per ragioni di semplicità concettuale.

Intuitivamente, si può ben capire come questo da una parte indebolisca di fatto la completezza del sistema reale, dall'altra introduca una problematica che deve essere risolta a un livello di astrazione più elevato.

3.3 Programmazione a oggetti in CSM

3.3.1 Caratteristiche

La programmazione contestuale estesa di CSM consente di catturare alcuni dei concetti fondamentali della programmazione a oggetti, quali quelli di oggetto, classe, gerarchia di classi, ereditarietà, scambio di messaggi, protezione dell'informazione.

In primo luogo, ciascuna unità può rappresentare un elemento del dominio applicativo, o meglio la sua parte caratteristica. La composizione delle unità in contesti consente infatti la definizione incrementale di oggetti, permettendo la specifica di unità che fungono da classi. In particolare, se $O \triangleleft C$ è il contesto associato con l'unità O, quest'ultima può essere letta come un'istanza della classe C, laddove O contenga i dati relativi all'oggetto $O \triangleleft C$, e C ne rappresenti la conoscenza comportamentale. A sua volta il contesto C, laddove composto da più unità, può essere letto come rappresentazione di una gerarchia di classi, legate tra loro da relazioni di tipo classe/superclasse. La relazione *super* tra teorie logiche, in particolare, può essere vista come la specializzazione di una relazione di tipo *isa* tra classi.

Si vede inoltre come un goal $O \leftarrow G$ possa essere letto come l'invio di un messaggio G all'oggetto O ($O \triangleleft C$). Le computazioni CSM possono essere quindi interpretate come una sequenza di messaggi scambiati tra entità computazionali distinte.

I meccanismi di binding contestuale consentono inoltre la realizzazione di meccanismi per l'ereditarietà. Ogni unità può ereditare conoscenza procedurale e dati dalle unità che la precedono in una gerarchia (in un contesto), e può fare riferimento a conoscenze di unità successive mediante, rispettivamente, la politica di binding eager e late, che riproducono i classici approcci *super* e *self* del linguaggi a oggetti.

Infine, la possibilità di associare una proprietà di visibilità a ciascun predicato di una teoria logica permette di realizzare information hiding: ciascuna proprietà di un oggetto, sia essa un metodo o un attributo, può essere resa accessibile o meno dall'esterno dell'oggetto, o da una classe all'altra all'interno di una tassonomia.

In aggiunta, la programmazione logica contestuale offre alcune caratteristiche originali che vanno ad arricchire lo spettro delle possibilità offerte da un linguaggio a oggetti.

Innanzitutto, la capacità di costruire dinamicamente contesti, tramite i meccanismi di estensione o la specifica esplicita nel cambio di contesto, consente di superare la tipica limitazione dei linguaggi imperativi a oggetti, che possono trattare solo gerarchie definite staticamente. In questo ambito, invece, è possibile combinare le unità logiche dinamicamente a formare qualunque gerarchia, cosicché risulta possibile per esempio costruire nuove classi, oppure associare la stessa istanza a classi diverse a tempo d'esecuzione.

Inoltre, il non determinismo proprio della programmazione logica, unito con la capacità di definire politiche di composizione di teorie che estendono o sovrascrivono predicati ridefiniti, aumenta la capacità di defizione incrementale degli oggetti. Infatti, le proprietà di una classe possono essere non solo ereditate (non definendo il predicato corrispondente) o ridefinite (ridefinendo il predicato e dichiarandolo overriding), ma anche *accresciute* (ridefinendo il predicato e dichiarandolo esteso) da una classe figlia, ampliandone di fatto la definizione.

3.3.2 Problemi

D'altra parte, questo approccio, come fatto rilevare anche nel capitolo 2, è gravato da pesanti limitazioni, che ne impediscono il riconoscimento come modello pienamente a oggetti e riducono lo spettro delle sue possibili applicazioni.

In primo luogo, classi e istanze esistono soltanto nell'interpretazione del programmatore: non c'è alcuna differenza concreta, a priori, tra un'unità che rappresenta un'istanza e una che rappresenta una classe. Non è quindi neppure pensabile che una 'classe' siffatta possa fungere da modello per una sua istanza.

Peraltro, l'insieme delle teorie logiche di un programma è staticamente definito nel testo del programma: non è quindi possibile creare alcuna istanza durante l'esecuzione del programma.

In aggiunta, manca completamente la nozione di stato di un oggetto, giacché ogni 'istanza' è configurata compiutamente all'interno del testo di un programma CSM. Di conseguenza, non c'è modo in cui un'oggetto possa riflettere lo stato di una computazione logica attraverso la sua configurazione di stato.

Ciò che serve è quindi un meccanismo per la generazione di nuove unità, in modo da consentire la creazione dinamica di istanze a partire dalle classi.

Così pure, si dovrà individuare un nuovo concetto di configurazione di un'unità, in cui la nozione di costruzione di una teoria logica possa coesistere con un modello computazionale di tipo dichiarativo.

3.4 Applicazioni di CSM

3.4.1 Programmazione contestuale e robotica

Parte non trascurabile del lavoro svolto nel triennio di ricerca è consistito nella validazione di CSM come strumento concreto per la risoluzione di problemi

reali. In particolare, si è rivolta l'attenzione all'analisi dell'impatto della programmazione logica contestuale estesa, e dell'ambiente di programmazione CSM su ambiti applicativi avanzati, che ne ponessero maggiormente in risalto pregi e limitazioni.

La scelta dell'ambito applicativo è caduta sulla robotica, sia per la sua rilevanza generale, sia perché propone le tematiche applicative le più diverse, svarianti dal *planning* classico (dove le capacità di elaborazione simbolica proprie dei linguaggi logici sono valorizzate) al controllo *real-time* di agenti di basso livello (dove, altresì, divengono palesi le carenze a livello di controllo dei linguaggi logici tradizionali).

In vista di tale sperimentazione, si è proceduto a un'integrazione del sistema CSM con l'ambiente APPEAL [APPE] di interfaccia SICStus/X-Window, in modo da creare un ambiente di sviluppo completo e maneggevole.

Si è potuto pertanto dimostrare come le estensioni proposte alla programmazione logica classica consentano effettivamente di affrontare con un certo successo aree applicative, come la robotica, usualmente ostiche per i linguaggi logici. In aggiunta, la carenza di ambienti di programmazione che sappiano rispondere globalmente alle esigenze molteplici ed eterogenee proprie dell'ambito robotico propone l'approccio logico contestuale esteso come un credibile candidato per il supporto dello sviluppo di sistemi e applicazioni robotiche.

Come è ovvio, tale sperimentazione ha proposto anche quegli stessi problemi già emersi dall'analisi teorica: l'incapacità di creare nuovi oggetti, di rappresentare lo stato di un oggetto, e quant'altro, sono limitazioni difficilmente sopportabili in ambito robotico.

Essendo comunque l'attività sperimentale ampiamente documentata, risulta preferibile non dilungarci sull'argomento, e rimandare piuttosto alle opportune pubblicazioni [DNOZ94a, DNOZ94b, NOZ93, ZCNO94].

3.4.2 Programmazione contestuale concorrente

Un'ulteriore esigenza nata dalla sperimentazione effettuata consiste nella capacità di gestire più processi computazionali indipendenti in modo concorrente. Sulla falsariga di CPU (descritto in [MeNa92]), quindi, è stata formulata una proposta di integrazione tra la programmazione logica contestuale e un modello per la comunicazione, coordinazione e sincronizzazione tra processi.

Tale modello [ODN95] è denominato *ACLT*, da Agents Communicating through Logic Theories, e si fonda su di un ambiente a teorie logiche multiple dove la teoria logica svolge la duplice funzione di astrazione di comunicazione e deposito di conoscenza condivisa.

L'integrazione concreta (in corso, [Ven95]) del modello ACLT in CSM potrà da una parte accrescere ampiamente lo spettro dei problemi affrontabili nella pratica dal sistema, dall'altra arricchire il modello consentendo la definizione di astrazioni di comunicazione assai più articolate e complesse di quelle proprie del modello originale.

3.5 Sommario

La programmazione logica contestuale consente di superare alcune delle limitazioni tipiche della programmazione logica classica, soprattutto nell'ambito dell'ingegneria del software.

Il modello CSM estende la programmazione contestuale verso la programmazione a oggetti, senza smarrire la sua caratterizzazione semantica di tipo dichiarativo. Le proprietà del modello comprendono alcune delle caratteristiche fondamentali dei linguaggi a oggetti, e ne ampliano per certi versi lo spettro.

La sua implementazione in SICStus Prolog ha consentito in primo luogo di verificare la realizzabilità del modello, e quindi di testarne l'efficienza ed efficacia in ambiti applicativi concreti quali la robotica.

Le limitazioni del modello, però, lasciano diversi problemi aperti che possono essere affrontati solo con l'ausilio di strumenti concettuali originali.

4 Abduzione e configurazione di oggetti

Questo capitolo si pone l'obbiettivo di mostrare quali astrazioni e concetti sono necessari per introdurre la nozione di stato nel framework logico a teorie multiple etichettate introdotto nel capitolo precedente. In particolare, si considererà un ambito privo della possibilità di composizione delle teorie, in cui cioè il contesto di prova per le formule etichettate è costituito da una singola unità e non da una sequenza di unità (ovvero un contesto propriamente detto). Ciò permette di introdurre in maniera semplice il concetto di *creazione* e *configurazione* di una teoria logica per abduzione, senza perdersi in dettagli relativi alla composizione di teorie, che propongono

problemi in qualche modo ortogonali a questi.

4.1 Configurazione di oggetti come costruzione di una teoria logica

Il primo problema da affrontare ora è quindi quello di trovare un modo per catturare soddisfacentemente la relazione classe/istanza in un ambito logico multi-teoria. La *creazione di un'istanza* è un punto fondamentale nella programmazione orientata agli oggetti, in cui è possibile generare dinamicamente nuovi oggetti durante una computazione. Essendo gli oggetti den nostro framework rappresentati da *unità*, (ossia, teorie aperte denotate da termini ground), si dovrà fornire un meccanismo che consenta l'introduzione di nuove unità (istanze) secondo la loro specifica di classe.

Una volta creata, una nuova istanza dev'essere propriamente configurata. Sono quindi necessari meccanismi non solo per la generazione di nuove istanze, ma anche per la *configurazione* (parziale o completa) di queste come risultato di una computazione logica. Di conseguenza, lo stato degli oggetti terrebbe effettivamente traccia dell'evoluzione della computazione, così come richiesto dal paradigma a oggetti.

In definitiva, ciò che serve è un concetto originale di configurazione di oggetti, che colleghi opportunamente un modello computazionale dichiarativo con la nozione di costruzione di teoria logica.

4.2 Abduzione

4.2.1 Cos'è l'abduzione?

Secondo Peirce [Pei58], l'*abduzione* è la forma di sintesi inferenziale che inferisce un caso particolare da (*i*) una regola generale, e da (*ii*) il risultato dell'applicazione della regola a tale caso. Da un punto di vista logico, se assumiamo la regola "*B* se *A*", e sappiamo che *B* è vero, potremmo allora

inferire (abdurre) *A*, dato che *A* insieme alla regola "*B* se *A*" implica logicamente *B*.

Più semplicemente, l'abduzione è una forma di *ragionamento all'indietro* (*backward reasoning*) dalle *osservazioni* alle *spiegazioni*. Se sappiamo che *A* una possibile precondizione per *B*, e osserviamo *B*, potremmo allora prendere *A* come la possibile spiegazione per *B*, usando la regola all'indietro. Naturalmente, la regola non "vale" all'indietro: "*B* se *A*" *non* implica "*A* se *B*". Per questo, una delle condizioni per assumere la verità di *A* nel caso sta nella consistenza di *A* con tutte le informazioni assunte in precedenza. La regola "*B* se *A*" viene quindi usata semplicemente quale suggerimento, che propone *A* come ipotesi esplicativa dell'osservazione *B*.

Quindi, quello abduttivo è una forma di *ragionamento ipotetico*, non potendo concludere *A* da "*B* se *A*" e *B*, mentre possiamo prendere *A* come ipotetica spiegazione per *B* secondo la regola "*B* se *A*". L'abduzione è di conseguenza intrinsecamente *annullabile*, poiché ogni ipotesi può essere rigettata ove non più consistente con nuove osservazione. Inoltre, dato che possono esistere più precondizioni diverse per *B* (i.e. "*B* se *A*" e "*B* se *C*") allo stesso tempo, l'abduzione è intrinsecamente *non monotona*.

Il classico esempio dell'erba bagnata può aiutare la comprensione dei punti enunciati sin qui.

Esempio 4.1.

Supponiamo valgano le seguenti regole:

erba-bagnata se *irrigatore-acceso*
erba-bagnata se *pioggia-ieri-notte*
scarpe-bagnate se *erba-bagnata*

Se osserviamo che abbiamo le

scarpe-bagnate

potremmo fare la supposizione che vi sia o l'*irrigatore-acceso* o che vi sia

stata *pioggia-ieri-notte*, oppure prendere entrambe le supposizioni come spiegazioni per l'osservazione. Se accettiamo per esempio che ci fosse l'*irrigatore-acceso*, e scopriamo poi che l'irrigatore non poteva che essere spento per qualche ragione (i.e. perché era rotto), dovremo rigettare l'ipotesi di *irrigatore-acceso*, e concludere che è caduta invece *pioggia-ieri-notte*.

L'abduzione è quindi lo strumento concettuale proprio per trattare *conoscenza incompleta*. Essa consente di assimilare *nuova informazione* consistentemente con quanto è gia noto (regole e osservazioni). Per esempio, nell'esempio 4.1, non abbiamo informazione sufficiente per determinare cosa sia effettivamente accaduto ieri notte. Se avessimo tutta l'informazione, non avremmo bisogno di cercare spiegazioni per il fatto che le scarpe sono bagnate. Poiché questo non è il nostro caso, assumiamo informazioni non disponibili prima su quanto è successo durante la notte.

4.2.2 Il framework abduttivo

Formalmente, un framework abduttivo può essere definito in termini deduttivi in qualunque sistema per la rappresentazione della conoscenza dotato della nozione di conseguenza logica ($\models$). Data una teoria T e un'osservazione G, una formula Δ è una *spiegazione abduttiva* per G se e solo se

(*i*) $T \cup \Delta \models G$

(*ii*) $T \cup \Delta$ è consistente

se, cioè, (*i*) la conoscenza aggiunta (Δ) a ciò che è già noto (T) risulta sufficiente a giustificare l'osservazione (G), e se (*ii*) l'aggiunta di nuova conoscenza preserva la consistenza della base di conoscenza ($T \cup \Delta$).

Naturalmente, l'abduzione ha senso solo quando la conoscenza disponibile non è sufficiente a fornire spiegazioni per le osservazioni, cioè quando non vale $T \models G$. Per esempio, le tre regole dell'esempio non bastano a concludere che *scarpe-bagnate* è vero. Inoltre, non tutte le possibili

spiegazioni sono ugualmente soddisfacenti: in effetti, potremmo altrimenti assumere G stesso come una spiegazione per G ($\Delta=G$) ogniqualvolta $T\cup G$ sia consistente, dato che varrebbe ovviamente $T\cup G \models G$. Per esempio, potremmo assumere *scarpe-bagnate* per spiegare l'osservazione *scarpe-bagnate* nell'esempio 4.1: è naturale che questa non sia però la scelta migliore, in quanto le ipotesi non sfruttano in alcun modo la conoscenza disponibile (le tre regole) per determinare qualche ragione per l'occorrenza delle osservazioni. Potremmo infine dire che la spiegazione *scarpe-bagnate* non spiega alcunché.

Potremmo d'altra parte assumere *erba-bagnata* come spiegazione di *scarpe-bagnate*. Tuttavia, tale ipotesi potrebbe non rispondere ai nostri requisiti per ciò che intendiamo essere una spiegazione. Potremmo perciò semplicemente *preferire* evitare di considerare *erba-bagnata* una spiegazione soddisfacente per *scarpe-bagnate*, poiché non spiega *a sufficienza*. Di conseguenza, un framework abduttivo dovrebbe consentire la discriminazione tra spiegazioni utili e non utili.

In effetti, ogni framework abduttivo definisce qualche criterio che consente di decidere se un dato Δ debba o meno essere accettato. Ciò corrisponde a stabilire dei confini intorno a quella parte specifica del dominio applicativo di cui abbiamo una conoscenza incompleta. Una volta delimitata *semanticamente*, tale parte può essere caraterizzata *sintatticamente*, in modo da restringere le spiegazioni a classi di formule prestabilite, dette *ipotesi abducibili*. Di conseguenza, l'insieme A delle ipotesi abducibili è il primo criterio sintattico per la selezione di spiegazioni accettabili per le osservazioni.

Esempio 4.2.

Date le regole dell'esempio 4.1, sia {*irrigatore-acceso*, *pioggia-ieri-notte*} l'insieme A delle ipotesi abducibili. Allora, né *scarpe-bagnate* né *erba-bagnata* possono più essere prese come spiegazioni accettabili per l'osservazione *scarpe-bagnate*.

Un'ulteriore restrizione tipica è quella che le ipotesi che possono essere tratte dall'insieme delle spiegazioni abducibili devono essere formule prive di variabili. Poiché l'abduzione ha a che fare con fatti che fungono sia da osservazioni sia da spiegazioni, appare naturale descriverli come formule ground, cosa che può comunque essere fatta senza perdita di generalità (un risultato più forte viene da [Poo88]), producendo allo stesso tempo un'ovvia semplificazione del carico computazionale corrispondente.

In aggiunta, sono di solito necessari criteri più raffinati per restringere le spiegazioni accettabili, per esempio per selezionare la spiegazione migliore tra una molteplicità di ipotesi disponibili. Per questo, è possibile definire un insieme *I* di *vincoli d'integrità* in un framework abduttivo, con lo scopo di ridurre il campo delle ipotesi abduttive. Questi vincoli vengono usati per definire criteri specifici per la verifica di consistenza delle ipotesi, o per determinare forme di preferenza tra differenti spiegazioni delle stesse osservazioni.

Example 4.3.

Date le regole dell'esempio 4.1, e le ipotesi abducibili dell'esempio 4.2, possiamo introdurre il seguente insieme di vincoli d'integrità *I*:

$\{\neg(\textit{irrigatore-rotto} \wedge \textit{irrigatore-acceso}),$
$\neg(\textit{trenta-gradi-ieri-notte} \wedge \textit{pioggia-ieri-notte})\}$

Se veniamo a sapere che c'erano *trenta-gradi-ieri-notte*, potremmo escludere la possibilità che ci sia stata *pioggia-ieri-notte*, assumendo nel contempo che ci sia *erba-bagnata* perché c'era l'*irrigatore-acceso*. D'altra parte, se invece osserviamo che la notte scorsa c'era l'*irrigatore-rotto*, dovremo rigettare l'ipotesi di *irrigatore-acceso*, assumendo invece che sia caduta *pioggia-ieri-notte*.

In definitiva, il framework abduttivo generale consiste in una tripla $\langle T, A, I \rangle$, in cui T è una teoria logica di qualche tipo, caratterizzata comunque da una nozione di conseguenza logica, A è un insieme di ipotesi abducibili, e I un insieme di vincoli d'integrità. Data tale tripla, Δ è una spiegazione della frase G se e solo se

(*i*) $T \cup \Delta \models G$

(*ii*) $T \cup \Delta$ soddisfa I [13]

premesso che Δ è un insieme di frasi (ground) costruite a partire dai predicati di A.

Il modello computazionale di un framework abduttivo è basato sulla nozione di osservazione come *goal*. Utilizzando una qualche forma di risoluzione (come la risoluzione lineare), una computazione cerca di trasformare un goal dato (che rappresenta un'osservazione da spiegare) in goal più semplici finché non riesce a ridurlo in termini di ipotesi accettabili.

Tali ipotesi devono poi essere controllate verificandone la consistenza rispetto ai vincoli d'integrità. Per questo, la tipica computazione abduttiva consiste di due passi: (*i*) generazione delle spiegazioni per le osservazioni date, e quindi (*ii*) verifica del fatto che la nuova base di conoscenza (la teoria e le nuove ipotesi) soddisfa i vincoli d'integrità.

4.2.3 Abduzione e programmazione logica

Se prendiamo T come un programma logico, A come $\{=\}$, e I come l'insieme dei vincoli della teoria dell'uguaglianza, ci troviamo ad aver ridefinito il framework logico classico come un framework abduttivo. I goal possono

---

[13] Diverse letture dei vincoli d'integrità conducono a differenti interpretazioni di cosa significhi qui "soddisfa" [KoSa87]. La *consistency view* richiede la consistenza globale di $T \cup \Delta \cup I$, una sorta di requisito "minimale". Alternativamente, la *theoremhood view* è più "forte", poiché richiede che $T \cup \Delta \models I$. A un livello intermedio si trova la *epistemic* (o *metalevel*) *view*, che guarda I come asserzioni di metalivello sulle proprietà della base di conoscenza ($T \cup \Delta$) piuttosto che sul dominio inteso.

essere visti come osservazioni, e le sostituzioni di risposta sono formule d'uguaglianza che fungono da spiegazioni abduttive per i goal. Si vede così come l'ambito concettuale della programmazione logica sia fondamentalmente abduttivo.

Più in generale, il framework dell' *Abductive Logic Programming* (ALP) [KKT92] estende il paradigma logico classico al supporto del ragionamento abduttivo. Un framework ALP può quindi essere descritto come una tripla $\langle P, A, I\rangle$, in cui P è un programma logico, A è un insieme di predicati abducibili , e I è un insieme di frasi della logica del prim'ordine che fungono da vincoli d'integrità.[14]

Il problema fondamentale da risolvere al fine di fornire una caratterizzazione semantica di un framework ALP è quello della definizione di un criterio in base al quale decidere se $P \cup \Delta$ soddisfa I. Una possibile soluzione è quella di scegliere la semantica S più appropriata per P, e definire poi una semantica per $\langle P, A, I\rangle$ in termini di S-modelli. Dato Δ, $M(\Delta)$ è un *S-modello generalizzato* per $\langle P, A, I\rangle$ se è un S-modello per $P \cup \Delta$, e $M(\Delta) \models I$.[15] Così, controllare l'accettabilità di una spiegazione Δ corrisponde a verificare l'esistenza di un S-modello $M(\Delta)$ che sia al contempo un modello per I.

4.2.4 Abduzione in un framework logico multiteoria

I predicati abducibili sono intrinsecamente *predicati aperti*: poiché rappresentano informazione incompleta, non si può applicare l'Assunzione di Mondo Chiuso (CWA) ai predicati di A. Questa nuova nozione di apertura si adatta naturalmente a un ambito logico multiteoria etichettato, a sua volta

---

14 Restringere i vincoli d'integrità a essere frasi del prim'ordine consente di esprimere la nozione di soddisfacibilità dei vincoli negli stessi termini di deduzione logica propria del paradigma logico.

15 Questo approccio promuove fondamentalmente l'interpretazione epistemica dei vincoli d'integrità, poiché guarda a essi come ad asserzioni di metalivello concernenti il *modello* della base di conoscenza piuttosto che non la base ($P \cup \Delta$) in sè.

fondato su un rilassamento della CWA.

Il framework abduttivo generale può essere facilmente modificato in modo da tenere in conto una molteplicità di teorie logiche. Intuitivamente, ciò corrisponde a partizionare la conoscenza incompleta (che vogliamo inferire abduttivamente) in parti corrispondenti agli elementi del dominio applicativo. Poiché le etichette delle teorie mappano le clausole di un programma in astrazioni del dominio inteso, la modifica si limita a etichettare i predicati abducibili, usando ovviamente i nomi delle teorie come etichette dei predicati. La nozione di *predicati abducibili etichettati* generalizza quello di predicati abducibili, cosicché le spiegazioni accettabili non sono più verità universali, bensì informazione correlata a un oggetto dato. Di conseguenza, ciascuna teoria rappresenta l'informazione relativa a un elemento del dominio inteso sia in termini di ciò che noto a priori sia in termini di ciò che può essere inferito durante una computazione.[16]

Così, un framework abduttivo etichettato può a sua volta essere descritto come una tripla $\langle P, A, I \rangle$, in cui P un programma logico a teorie multiple etichettate (cioè, un insieme di teorie logiche etichettate), A è un insieme di abducibili etichettati (cioè, simboli di predicato etichettati), e I è l'insieme di vincoli d'integrità (costruiti a partire da formule etichettate). Anche le considerazioni di carattere semantico possono essere facilmente generalizzate a partire dal caso monoteoria, per cui non ci soffermeremo su di esse in questa sede. L'esempio che segue può consentire la comprensione di alcune idee di base.

Esempio 4.4.

Il nostro dominio applicativo consiste in tre gatti, Kitty, Tiger, e Jojo.

---

[16] [BLMM91] è ancora fondato sulla nozione di verità universale, pur se discute l'abduzione in un ambito multiteoria (non etichettato). Tuttavia, molti dei concetti ivi introdotti possono essere adattati a un framework etichettato, e verranno quindi ampiamente utilizzati nel seguito.

Sappiamo che Kitty è di colore bianco, Tiger è un maschio nero, e Jojo è di colore rosso. Il sesso di Kitty e Jojo non è noto. In aggiunta, conosciamo alcune regole che si applicano ai gatti: (*i*) i maschi amano le femmine, e le femmine amano i maschi; (*ii*) un maschio odia gli altri maschi, così come una femmina odia le altre femmine.

Le teorie etichettate che seguono costituiscono un programma logico multiteoria che può essere sfruttato per rappresentare in forma clausale le conoscenze date:

```
kitty                tiger                jojo
  colour(white).       colour(black).       colour(red).
                       sex(male).

catRules
  likes(CatA,CatB) :- CatA:sex(male), CatB:sex(female).
  likes(CatA,CatB) :- CatA:sex(female), CatB:sex(male).
  hates(CatA,CatB) :- CatA:sex(male), CatB:sex(male).
  hates(CatA,CatB) :- CatA:sex(female), CatB:sex(female).
```

Dobbiamo cercare di capire il sesso di Kitty e Jojo. Di conseguenza, l'insieme *A* dei predicati abducibili etichettati è {`kitty:sex/1, jojo:sex/1`}.

Se osserviamo ora che Kitty ama Jojo, potremmo sfruttare le regole sui gatti per ottenere una spiegazione. Da `catRules:likes(kitty,jojo)` potremo così inferire che o Kitty è un maschio e Jojo una femmina (`kitty:sex(male)`, `jojo:sex(female)`), oppure che, viceversa, Kitty è una femmina e Jojo un maschio (`kitty:sex(female)`, `jojo:sex(male)`). Nel caso poi che un'ulteriore osservazione ci indichi che Jojo odia Tiger, potremmo spiegare `catRules:hates(jojo,tiger)` assumendo (dalla terza regola di `catRules`) che Jojo è un maschio (`jojo:sex(male)`), ben sapendo che Tiger è un maschio (e che non possiamo fare nuove ipotesi sul suo sesso, poiché `tiger:sex/1`$\notin A$).

Allo scopo di evitare l'inconsistenza semantica legata al contemporaneo essere Jojo un maschio e una femmina, dovremo introdurre almeno un vincolo d'integrità che asserisca che non è ammissibile (per un gatto) essere un maschio e una femmina contemporaneamente. Definiremo quindi *I* come

I = {∀X ¬(X:sex(male) ∧ X:sex(female))}

Così, jojo:sex(male) e jojo:sex(female) non possono fungere da ipotesi contemporaneamente. Quindi, l'unica spiegazione disponibile per entrambe le osservazioni è che Kitty è una femmina e Jojo un maschio. In definitiva, data la tripla

P = {kitty, tiger, jojo, catRules}
A = {kitty:sex/1, jojo:sex/1}
I = {∀X ¬(X:sex(male) ∧ X:sex(female))}

e le osservazioni

G = {catRules:likes(kitty,jojo),
catRules:hates(jojo,tiger)}

possiamo assumere

Δ = {kitty:sex(female), jojo:sex(male)}

come spiegazione per G poiché abbiamo che vale

(*i*) $P \cup \Delta \models G$

(*ii*) $P \cup \Delta$ soddisfa I

Infatti, utilizzando il modello minimo di Herbrand come l'*S*-model naturale per la nostra caratterizzazione semantica, $M(\Delta)$ consiste nei seguenti atomi etichettati:

$M(\Delta)$ = {
kitty:colour(white), kitty:sex(female),
jojo:colour(red), jojo:sex(male),
tiger:colour(black), tiger:sex(male),
catRules:likes(kitty,jojo),
catRules:likes(kitty,tiger),
catRules:likes(jojo,kitty),
catRules:likes(tiger,kitty),
catRules:hates(jojo,tiger),
catRules:hates(tiger,jojo)
}

Poiché $G \subset M(\Delta)$, e $M(\Delta)$ è per definizione il modello minimo di Herbrand

$P \cup \Delta$, ne deriva che $P \cup \Delta \models G$. Inoltre, poiché $M(\Delta) \models I$, possiamo asserire che $P \cup \Delta$ soddisfa I.

4.3 Configurazione dello stato di un oggetto

4.3.1 Configurazione di oggetti

Le classi sono *template* per gli oggetti. Una volta creato, un oggetto di una data classe in un ambito orientato agli oggetti è concettualmente uno spazio vuoto ma strutturato in modo da poter essere completato con i dati relativi all'oggetto, utilizzando i metodi della classe corrispondente. La *configurazione di un oggetto* è quindi il processo con cui si assegnano valori agli attributi di un oggetto, a partire da uno stato iniziale *nullo*.

Questa operazione viene usualmente detta *inizializzazione* nei linguaggi a oggetti imperativi, che, adottando un approccio basato su funzioni e assegnamento, non sono in grado di trattare oggetti il cui stato sia solo parzialmente specificato. Per questo, tali linguaggi, dovendo computare solo su oggetti completamente configurati, forniscono strumenti (come la nozione di *costruttore* del C++) per rendere l'inizializzazione un'operazione atomica a livello di linguaggio. La configurazione degli oggetti non è dunque percepita come questione di rilevanza alcuna nei linguaggi orientati agli oggetti basati sul tradizionale paradigma imperativo.

D'altra parte, i linguaggi logici promuovono una visione decisamente diversa della nozione di configurazione di un oggetto. Punto essenziale nel progetto di un programma logico è la costruzione dei dati in risposta a dimostrazioni di successo. Poiché il paradigma logico è basato su relazioni e unificazione, gli oggetti parzialmente specificati rivestono un ruolo fondamentale in ogni computazione logica, ragion per cui, a differenza dei linguaggi imperativi, la configurazione di un oggetto è un punto focale per un

linguaggio logico orientato agli oggetti, come mostrato nel capitolo 2.

Inoltre, un linguaggio che fornisca strumenti per la configurazione dei dati di un'istanza fornisce effettivamente una nozione di *stato* di un oggetto, anche senza dover trattare di *modifica* dello stato. Infatti, secondo [Weg87], lo stato di un oggetto è ciò che conserva la memoria della computazione; inoltre, il risultato di un'invocazione del metodo di un oggetto dipende effettivamente dalla storia della computazione, e non è invece predeterminato dalla definizione dell'oggetto stesso (come nel caso degli "oggetti funzionali"). Possiamo pertanto asserire che la configurazione di un oggetto è a tutti gli effetti configurazione dello stato di un oggetto, e che un linguaggio logico che supporti tale nozione è un linguaggio a oggetti a pieno titolo.

Trattare la configurazione di oggetti è trattare conoscenza incompleta. La struttura della conoscenza correlata a un'istanza è data dal modello della classe: ciò che dev'essere stabilito è la collezione dei valori da associare agli attributi di un oggetto. Da questo punto di vista, una computazione che determini incrementalmente i dati di un oggetto può a tutti gli effetti essere interpretata come un processo di assimilazione di nuova conoscenza volto al completamento dell'informazione correlata all'oggetto stesso.

Più specificamente, in un framework logico (in cui gli oggetti siano rappresentati da teorie etichettate, e tanto i metodi quanto gli attributi siano clausole) il processo di configurazione di un oggetto consiste nell'assimilazione di nuovi assiomi, che rappresentano i dati dell'oggetto. La conoscenza delle teorie logiche rappresentanti le istanze è incompleta, e dev'essere completata come risultato di una computazione logica: un linguaggio logico orientato agli oggetti deve quindi consentire di associare a ogni teoria-istanza un "template di assiomi" da istanziare poi con un processo di assimilazione di nuova conoscenza che preservi la lettura dichiarativa della programmazione logica.

4.3.2 Abduzione e configurazione di oggetti

Una volta data l'idea dello stato di un oggetto come informazione incompleta, l'uso dell'abduzione come strumento concettuale per la configurazione dichiarativa dello stato di un oggetto segue in modo del tutto naturale. In particolare, essendo lo stato di un oggetto usualmente strutturato in termini di attributi, mentre l'informazione incompleta è sintatticamente definita in termini di abducibili, la soluzione più ovvia è quella di concepire gli attributi di un oggetto come abducibili. Per esempio, i gatti dell'esempio 4.4 possono essere visti come oggetti che hanno attributi come *sesso* e *colore*.

In particolare, l'uso dei vincoli d'integrità nell'esempio sopra richiama la tipica percezione degli attributi nei linguaggi a oggetti: gli attributi sono cioè qualificatori univoci, per i quali non ha senso avere più di un valore ciascuno. Nell'esempio, ciascun gatto è qualificato da uno e un solo attributo *sesso*, e i vincoli d'integrità assicurano la soddisfazione di tale condizione.

Gli attributi di un oggetto possono pertanto essere facilmente modellati associando un vincolo d'integrità di *singolarità* a un sottinsieme (eventualmente improprio) dei predicati abducibili, detto l'insieme dei predicati *abducibili singoli*. Se p è un abducibile singolo, aggiungiamo allora

$$\forall t, t' \, (p(t) \wedge p(t') \Rightarrow t = t') \qquad (4.1)$$

all'insieme dei vincoli d'integrità.

La nozione di abducibile singolo può essere agevolmente estesa all'ambito multiteoria etichettato, dove va a corrispondere direttamente alla nozione di attributo (monovalore) di un oggetto. Se O è una teoria logica, e p un suo singolo abducibile, la seguente formula va a fare parte dell'insieme I dei vincoli d'integrità::

$$\forall t, t' \, (O{:}p(t) \wedge O{:}p(t') \Rightarrow t = t') \qquad (4.2)$$

L'esempio 4.5 mostra come l'esempio 4.4 può essere reinterpretato alla luce di questa prospettiva a oggetti.

Esempio 4.5.

Il dominio applicativo dell'esempio 4.4 può essere connotato come un sistema di diversi oggetti. I tre gatti Kitty, Tiger, e Jojo sono rappresentati come tre oggetti aventi sesso e colore come loro attributi. Conosciamo il colore di tutti e tre (Kitty è di colore bianco, Tiger nero, e Jojo rosso), ma solo il sesso di Tiger (che è un maschio), mentre quello di Kitty e Jojo è ignoto. In aggiunta, abbiamo informazioni relative alla classe dei gatti, rappresentate in una quarta teoria contenente alcune regole riguardanti le preferenze sessuali feline.

Quindi, l'insieme degli attributi è costituito da fatti logici (i valori noti) oppure da abducibili, delimitati da specifici vincoli d'integrità. La formula (4.2) si traduce in questo caso in due vincoli:

$\forall$`t, t' kitty:sex(t)` $\wedge$ `kitty:sex(t')` $\Rightarrow$ `t = t'`
$\forall$`t, t' jojo:sex(t)` $\wedge$ `jojo:sex(t')` $\Rightarrow$ `t = t'`

Vincolando poi il nostro sistema con le relazioni

```
catRules:likes(kitty,jojo), catRules:hates(jojo,tiger)
```

questo si va a configurare in modo da soddisfare i vincoli dati. Essendovi una sola configurazione disponibile che soddisfi i vincoli (quella con `kitty:sex(female)` e `jojo:sex(male)`), l'attributo `sex/1` degli oggetti `kitty` e `jojo` viene configurato conseguentemente.

4.4 Creazione di oggetti

4.4.1 Creazione di teorie

La *creazione* di oggetti è l'altra questione fondamentale che un linguaggio logico che voglia dirsi orientato agli oggetti deve affrontare. Rappresentare gli oggetti come teorie logiche trasforma il problema della creazione di nuovi oggetti a quello di consentire la creazione di nuove teorie in risposta a computazioni logiche.

Dato che lo scopo principale di questo lavoro è la definizione di un modello per un linguaggio logico a oggetti che preservi la lettura dichiarativa, il problema è se sia possibile creare teorie logiche in modo dichiarativo, e, se sì, come.

Dal punto di vista di un programma, questo tema può essere visto come un'estensione del problema della configurazione di teorie: là, alcune teorie logiche *date* (staticamente) dovevano essere configurate propriamente associando a esse nuovi assiomi; qui, invece, abbiamo a che fare con teorie completamente *nuove*, ossia non definite in alcun modo nel testo del programma. In effetti, abbiamo a che fare con una sorta di *estensione di programma* più radicale, in cui solo un numero limitato di teorie componenti il nostro database logico è dato testualmente, mentre le altre sono generate dinamicamente durante le computazioni. Distingueremo allora la parte statica del programma (che diremo *core*) da quella dinamica (che diremo *estensione*).

4.4.2 Estensione di programma

Nel framework multiteoria etichettato di base, assumiamo che la definizione testuale di un programma P contenga già tutte le teorie che possono essere utilizzate durante una computazione: poiché esse appartengono alla parte core del programma, le diciamo *teorie core* di P. Questa assunzione implicita corrisponde a limitare il *range* degli elementi di H_P che possono essere usati per denotare una teoria all'insieme dei *nomi core*, ossia quei termini ground che denotano le teorie core.

La conseguenza primaria di questo approccio è che quando si incontra una formula $O{:}G$, questa viene considerata falsa per difetto ogniqualvolta O non sia un nome core. In qualche modo, benché l'assunzione di mondo chiuso sia stata abbandonata nell'ambito ristretto della singola teoria logica, è stata implicitamente reintrodotta nell'ambito più ampio del programma, ragion per cui la ridefiniamo come *assunzione di programma chiuso* (CPA).

Poiché, d'altra parte, stiamo cercando una forma di creazione (implicita o

esplicita) di teorie, ove né la teoria né la sua denotazione sia data nel testo del programma, risulta necessario un approccio differente. In particolare, anche la CPA dovrebbe essere abbandonata per i programmi etichettati come la CWA lo è stata per le teorie etichettate, e si dovranno esplorare le possibilità e le conseguenze di un'*assunzione di programma aperto* (OPA): come possiamo estendere un programma logico dichiarativamente?

4.4.3 Estensione di programma per abduzione

Possiamo dividere il problema della creazione dichiarativa di nuove teorie logiche in due sottoproblemi distinti:

1) come introdurre nuovi identificatori?

2) come associare nuovi assiomi ai nuovi identificatori?

L'OPA rilascia il vincolo che solo le teorie core possono essere utilizzate in un programma multiteoria per la dimostrazione delle formule. Dal punto di vista sintattico, questo corrisponde all'abbandono della restrizione sugli identificatori: *tutti* i termini ground sono allora da considerarsi nomi di teoria. Così, qualunque messaggio del tipo O:G, ove O appartenga al dominio H_P, non può essere rigettato finche non siano noti gli assiomi che compongono O.

Poiché ora H_P equivale all'insieme degli identificatori di teorie, e rappresenta nel contempo gli oggetti del dominio applicativo, ogni oggetto di tale dominio è in linea di principio rappresentato da una teoria logica. In particolare, le teorie core definite nel testo del programma rappresentano solo una parte limitata del dominio inteso.

Se ora denotiamo l'insieme dei nomi core di P come $core_P$, allora

$$extension_P ::= H_P - core_P \qquad (4.3)$$

è l'insieme dei termini ground che possono fungere da identificatori per le teorie logiche create dinamicamente, dette *teorie estensione* poiché appartengono alla parte estensione del programma. Coerentemente,

extension$_P$ è l'insieme dei *nomi estensione* del programma.

Il primo risultato è che gli identificatori delle teorie estensione non devono essere *introdotti* con apposite dichiarazioni o altro, ma semplicemente *usati*, poiché sono implicitamente definiti dal programma stesso. Ciò che deve allora essere determinato è invece quali clausole costituiscono la teoria associata a un termine ground utilizzato come nome estensione. L'idea più ovvia è quella di utilizzare ancora l'abduzione, dato che l'associazione di assiomi a nomi di teoria è ancora una volta un problema di configurazione di teorie. Il problema è come adattare il framework abduttivo, laddove sarebbero necessaria la capacità di esprimere a priori un'infinità di abducibili e di vincoli d'integrità, che comunque non potrebbe essere nota e definita staticamente.

Di conseguenza, l'approccio più semplice sarebbe quello di concepire il framework abduttivo come un'entità non completamente definita a priori, ma che può crescere ed evolvere con la computazione. In questo modo, si potrebbe consentire a una computazione di introdurre (implicitamente o esplicitamente) nuovi abducibili e nuovi vincoli d'integrità, eliminando i problemi della definizione statica, finita e a priori.

In un ambito logico monoteoria (il programma è una unica teoria logica) questa opzione introdurrebbe gravi problemi di coerenza. Infatti, la definizione dell'insieme degli abducibili determina una partizione dell'insieme dei predicati in abducibili (definiti esplicitamente) e non abducibili (definiti implicitamente per complemento). Aggiungere dinamicamente la qualifica di abducibile a un predicato *modifica* a tutti gli effetti il programma logico e la sua consistenza d'insieme.

In un framework logico multiteoria, tutto cambia: ossia, l'osservazione precedenti vale solo per le teorie specificate staticamente (le teorie core), ma non per le altre, di cui nulla è noto a priori. Inoltre, fintanto che la dimostrazione si svolge in una teoria, le ipotesi sulle altre non interessano, e non inficiano la coerenza del sistema.

Il punto sta quindi nel confine costituito dall'utilizzo di una teoria

estensione in un programma. Se le proprietà del framework abduttivo rispetto alle teorie core sono fissate staticamente una volta per tutte, e si è invece in grado di esprimere *atomicamente* abducibili e vincoli d'integrità per le teorie estensione, si può pensare di lavorare in un framework abduttivo *estendibile* senza problemi di coerenza. L'atomicita' (a livello di linguaggio) consente infatti di passare da un framework in cui nulla è noto di una data teoria estensione (in cui quindi tutto è ammissibile) a una sua estensione in cui sono noti tutti gli abducibili e i vincoli della teoria stessa, ed è quindi possibile determinare l'ammissibilità di ogni operazione.

Dato quindi un nome estensione, devo essere in grado di specificare atomicamente i suoi abducibili (accrescendo quindi l'insieme *A* del framework abduttivo) e gli eventuali vincoli d'integrità realtivi a essi (accrescendo *I*). Per esempio, un linguaggio potrebbe essere dotato di un apposito metapredicato

```
singleAbducible(ET, PredList)
```

con cui alla teoria estensione `ET` vengono associati una serie di predicati abducibili (`PredList`), e per ciascuno sono definiti i vincoli d'integrità di singolarità (derivati dalla formula (4.1)).

Esempio 4.6.

Si riprenda l'esempio 4.5, e si supponga di voler introdurre dinamicamente un nuovo gatto, Trudy, il cui colore e sesso non sono noti, potendo per esempio sfruttare il metapredicato `singleAbducible` mostrato sopra. In particolare, la seguente metarelazione

```
singleAbducible(trudy,[sex/1,colour/1])
```

aggiunge `trudy:sex/1` e `trudy:colour/1` all'insieme *A* degli abducibili, e accresce nel contempo l'insieme dei vincoli d'integrità *I* con i vincoli di singolarità

```
∀t, t' trudy:sex(t) ∧ trudy:sex(t') ⇒ t = t'
∀t, t' trudy:colour(t) ∧ trudy:colour(t') ⇒ t = t'
```

Se alle conoscenze dell'esempio 4.5 aggiungiamo ora che Trudy ha lo stesso colore di Kitty ma la odia,

```
catRules:likes(kitty,jojo), catRules:hates(jojo,tiger),
catRules:hates(trudy,kitty),
kitty:colour(C), trudy:colour(C)
```

il sistema inferirà, olte al sesso di Kitty e Jojo, anche colore e sesso di Trudy, abducendo i seguenti assiomi:

```
kitty:sex(female)
jojo:sex(male)
trudy:sex(female)
trudy:colour(white)
```

4.5 Un possibile approccio per la semantica dichiarativa

4.5.1 Modelli ammissibili per un programma con OPA

Una possibile connotazione dichiarativa del modello presentato nelle sezioni precedenti può prendere le mosse dalla caratterizzazione dichiarativa dei linguaggi logici a teorie multiple etichettate, basata sulla nozione di modello ammissibile (si veda l'appendice B).

Infatti, i modelli di Herbrand ammissibili si fondano su ipotesi della forma *T*:*G*: estendendo le possibili ipotesi anche a teorie che non sono dichiarate staticamente nel testo di *P*, ossia rilasciando ogni vincolo su *T* che non sia l'appartenenza a H_P, possiamo costruire modelli ammissibili per tutte le teorie core anche nel caso di OPA.

Denotiamo allora con *CoreOpen*(*T*) l'insieme le formule *O*:*G* di *Open*(*T*) (che diremo formule *coreopen*) in cui *O* è una teoria core, e *ExtOpen*(*T*) tutte le altre formule di *Open*(*T*) (che diremo formule *extension-open*), in cui ovviamente *O* è una teoria estensione. Definiamo poi con significato ovvio gli insiemi *CoreOpen*(*P*) e *ExtOpen*(*P*) come unione degli insiemi corrispondenti

relativi alle teorie core di *P*.

Non è quindi necessario ridefinire la nozione di modello ammissibile per una teoria, anche se resta il problema di come determinare il modello ammissibile di una teoria estensione con l'assunzione OPA, oppure, dualmente, come risolvere le ipotesi relative alle teorie estensione in fase di composizione.

Il modo più semplice di affrontare il problema si basa sull'idea che i goal external-open sono in linea di principio "esterni" al programma definito staticamente, e che quindi la caratterizzazione di un programma può giovarsi dello stesso approccio dei modelli ammissibili usato per le sue teorie, sfruttando come ipotesi le formule external-open. Diciamo allora *ipotesi di estensione* ogni sottoinsieme (proprio o improprio) di $ExtOpen(P)$

Riprendendo quindi la nozione di modello ammissibile per una teoria T, e quella di operatore di composizione ϕ_P (definizioni B.1 e B.2 dell'appendice B), modificata a comprendere il passaggio all'OPA sostituendo la condizione $T \in P$ con $T \in H_P$ (che ne preserva le proprietà di monotonicità e continuità delle proposizioni B1 e B2),

$$\phi_P ::= \lambda I\ I \cup \{T{:}A \mid T \in H_P \wedge A \in M_T(H) \wedge H \subseteq I\}$$

possiamo ora definire il concetto di *modello di Herbrand ammissibile* per un programma *P*.

Definizione 4.1. *Modello di Herbrand ammissibile per un programma.*

Sia *P* un programma a teorie multiple etichettate, e sia $H \subseteq ExtOpen(P)$ un insieme di ipotesi di estensione per *P*. Diciamo allore *modello ammissibile di Herbrand* per il programma *P* secondo le ipotesi d'estensione *H* l'insieme ottenuto applicando $\phi_P \uparrow \omega$ ad *H*, e lo denotiamo con $||P||(H)$. Formalmente,

$$||P||(H) ::= \phi_P \uparrow \omega(H)$$

Applicando l'operatore $\phi_P \uparrow \omega$ a diverse ipotesi d'estensione otteniamo ovviemante diversi modelli ammissibili per lo stesso programma. In particolare, la CPA corrisponde all'ipotesi di estensione nulla ($H=\emptyset$), mentre l'OPA "totale" (tutto ciò che è riguarda le teorie estensione è vero) corrisponde all'ipotesi d'estensione massima ($H=ExtOpen(P)$).

Dato che il criterio di minimalità non può più essere utilizzato in maniera significativa (esso corrisponde a $\phi_P \uparrow \omega(\emptyset)$, ossia al caso di CPA), la questione dell'unicità del modello per un programma *P* si riduce a capire se esista o meno *un'unica* ipotesi d'estensione $H \subseteq ExtOpen(P)$ accettabile per il programma. La questione principale diviene quindi innanzitutto quella di stabilire se esistano criteri che consentano di selezionare le possibili ipotesi d'estensione.

La soluzione a questo problema è quella di introdurre una nozione di *consistenza*, che consenta di determinare l'accettabilità di un'ipotesi d'estensione. Tramite l'introduzione di metapredicati che definiscano gli abducibili per una teoria estensione, o che definiscano nuovi vincoli d'integrità (come il metapredicato `singleAbducible` visto sopra), è possibile escludere possibili ipotesi d'estensione. Per esempio, l'esempio 4.6 mostra come è possibile escludere ipotesi del tipo {`trudy:sex(female)`, `trudy:sex(male)`, `colour(white)`, `color(black)`} tramite una dichiarazione del tipo

```
singleAbducible(trudy,[sex/1,colour/1])
```

Ogni linguaggio che adotti questo approccio dovrà quindi adottare una precisa politica nella gestione delle ipotesi d'estensione, poiché nella loro disciplina sta la utilità pratica del modello: ogni linguaggio dovrà quindi fornire gli strumenti (linguistici e metalinguistici) volti a questo scopo. In ogni caso, ciò che emerge da questa discussione è che la questione dell'unicità non è più così centrale come nella programmazione logica classica.

4.6 Sommario

Il problema della creazione e configurazione di oggetti in un ambito logico multiteoria etichettato si riconduce alla ricerca di un meccanismo per la creazione e la configurazione dichiarativa di teorie logiche.

L'abduzione costituisce lo strumento concettuale per la manipolazione della conoscenza incompleta: una volta riconosciuta la configurazione di un oggetto rimanda all'incompletezza dell'informazione su di esso, risulta naturale utilizzare un framework abduttivo per la configurazione dichiarativa di teorie logiche.

Estendendo poi la nozione di framework abduttivo, e abbandonando contemporaneamente l'assunzione di programma chiuso (ossia fissato staticamente nel testo del programma), si può catturare anche il concetto di creazione dinamica di teorie logiche in modo dichiarativo.

Ciò che resta da fare è quindi di estendere il modello concettuale qui definito in un framework privo di composizione di teorie all'ambito contestuale completo, e di stabilire un modello computazionale coerente.

5 Vincoli di metalivello e modello computazionale

5.1 Vincoli di dimostrabilità

Una volta definito il framework logico abduttivo per la creazione e configurazione dinamica di teorie logiche, è necessario fornire il modello delle computazioni, così che possa fungere da archetipo per la definizione di veri e propri linguaggi logici orientati agli oggetti.

La soluzione scelta è quella di prendere come riferimento il modello contestuale esteso descritto nel capitolo 3, integrandolo con la nozione di estensione di programma logico, la quale va precisamente a coprire le principali lacune del modello evidenziate nellla sottosezione 3.3.2.

Non è peraltro obbiettivo di questo lavoro descrivere un particolare linguaggio con una precisa caratterizzazione operazionale e dichiarativa, bensì organizzare lo schema concettuale in cui possano essere definite *classi* di linguaggi logici orientati agli oggetti. Per questo, ci limiteremo in questo capitolo a delineare le idee-guida di tale schema, evitando di inoltrarci sul sentiero delle scelte linguistiche e ingegneristiche che poco hanno a che fare col modello.

Solo per ragioni di esemplificazione e di concretezza, infine, si istanzierà il modello a un semplice linguaggio (che costituisce una versione semplificata di quello attualmente in fase di implementazione sul supporto CSM) con cui saranno illustrati pochi, ma significativi esempi.

5.1.1 L'idea

L'idea fondamentale per la definizione di un modello computazionale logico orientato agli oggetti nasce dall'osservazione fatta a conclusione della sottosezione 3.2.5, dove si faceva notare come per esempio la non completa istanziazione dell'identificatore di una teoria in un'estensione di contesto non fosse gestibile in maniera soddisfacente dal sistema.

Generalizzando al caso presente, in cui dobbiamo introdurre, in ambito logico contestuale, l'estensione di programma, potremo dire che la strategia di ricerca tipica del Prolog e delle sue estensioni non consente di trattare la dimostrazione di una formula in un contesto di prova non completamente specificato. Ciò comprende sia il caso di identificatore non specificato completamente, sia il caso di dimostrazione in una teoria estensione, di cui ancora non sia tutto noto.

Dal punto di vista a oggetti, ciò equivale a dire che la regola di calcolo

usuale non consente di trattare oggetti non completamente specificati, perdendo così uno dei maggiori vantaggi della programmazione logica, come discusso nel capitolo 2.

Ciò che si vorrebbe ottenere, invece, è una completa *neutralità* della dimostrazione, in cui non solo una formula atomica può essere configurata in risposta a una sua dimostrazione in un contesto di prova dato, ma anche un contesto di prova può essere configurato in risposta alla dimostrazione di una formula atomica data. L'idea fondamentale è quindi quella di considerare i goal aperti come *relazioni di metalivello* tra formule atomiche e contesti di prova, che *vincolano* tanto il contesto quanto la formula.

Un goal aperto va quindi interpretato come un *vincolo di dimostrabilità* tra un contesto e una formula atomica, per soddisfare il quale il contesto è vincolato a implicare logicamente la formula (e viceversa). A tale scopo, tanto la formula quanto il contesto possono essere soggetti a configurazione, per cui, dualmente, la valutazione del goal non richiede che il contesto (né la formula, ovviamente) sia completamente specificato perché la computazione possa avere luogo.

5.1.2 Il modello computazionale

Il modello computazionale corrispondente non può che essere di tipo *data-driven*. Infatti, un vincolo di dimostrabilità può essere risolto solo tramite un'effettiva computazione che provi una formula in un contesto dato, cosa che può avvenire, per gli strumenti che abbiamo a disposizione, solo quando si sia accumulata informazione sufficiente a determinare (o configurare) il contesto di prova. Adottando per i vincoli di dimostrabilità il tipico approccio della *programmazione a vincoli* è facile delineare tale modello.

Possiamo quindi interpretare il testo di un programma come una serie di vincoli e *metavincoli* che costituiscono la parte immutabile dello *store* della computazione. Per esempio, faranno parte dello store l'informazione sui nomi core, l'appartenenza delle clausole a una data teoria core, le relazione *super* tra

le teorie core.

La computazione logica, poi, accrescerà incrementalmente lo store durante la sua evoluzione, aggiungendo nuovi (meta)vincoli, rappresentanti per esempio gli abducibili delle teorie estensione, o i vincoli d'integrità. Per questo scopo, ogni linguaggio che realizzi questo modello dovrà fornire appositi (meta)predicati, come già discusso nel precedente capitolo.

Quindi, la semantica operazionale corrispondente potrà essere data nel modo caratteristico dei linguaggi a vincoli, ossia tramite un sistema a stati, in cui uno stato è rappresentato da una tripla $\langle A, C, S \rangle$, in cui A sia il risolvente, C ed S gli store dei vincoli attivi e passivi (si veda [JaMa94] per maggiori dettagli).

In questi sistemi, il passaggio da uno stato all'altro avviene mediante quattro regole di transizione, che corrispondono a quattro fasi concettualmente distinte delle computazioni a vincoli, e che possiamo ritrovare ovviamente anche nel nostro caso. Un linguaggio, per essere definito compiutamente, dovrà poi stabilire anche una regola di calcolo che consenta, di volta in volta, la selezione dell'opportuna regola di transizione da applicare allo stato corrente.

La prima regola, la transizione di *inferenza*, consente di trasformare lo store dei vincoli (attivi e passivi) estraendo l'informazione contenuta implicitamente nei vincoli. Per esempio, una computazione contestuale potrebbe ricavare l'informazione relativa agli abducibili di una data teoria estensione, coinvolta in un processo di dimostrazione.

La seconda, transizione di *risoluzione*, rappresenta il passo di risoluzione logica classico, che in questo caso va semplicemente esteso a comprendere il binding contestuale.

La terza, transizione di *vincolamento*, consente l'introduzione di nuovi vincoli nello store. La computazione logica, infatti, accrescerà incrementalmente lo store durante la sua evoluzione, aggiungendo nuovi metavincoli, rappresentanti per esempio gli abducibili delle teorie estensione,

o vincoli d'integrità. Per questo scopo, ogni linguaggio che realizzi questo modello dovrà fornire appositi metapredicati, come già discusso nel precedente capitolo.

La quarta, infine, transizione di *consistenza*, non effettua alcun cambiamento di stato, bensì si limita a verificare la consistenza dello stato dei vincoli. In particolare, la nozione di consistenza sarà ovviamente fondata sul framwork concettuale abduttivo definito nel precedente capitolo. Non si potrà, per esempio, accettare che nuovi abducibili siano definiti per una teoria core.

Ovviamente, come già per il modello concettuale abduttivo, questo approccio fornisce un semplice *archetipo* per un modello computazionale di linguaggio logico orientato agli oggetti. Ciascun linguaggio, poi, dovrà precisare in dettaglio i vincoli ammessi, i metapredicati corrispondenti, la nozione di consistenza, la funzione di inferenza, la regola di calcolo. Per un esempio concreto di semantica operazionale, si veda il semplice esempio riportato in [OmNa94].

5.2 Classi e istanze come teorie logiche

5.2.1 Un possibile linguaggio

Un esempio di linguaggio logico orientato agli oggetti che realizzi i modelli presentati in questo e nel precedente capitolo può essere costruito estendendo il framework CSM del capitolo 3. In particolare, modificheremo la regola di calcolo in senso data-driven secondo quanto visto nella sezione precedente, in modo del tutto intuitivo, adottando nel contempo l'assunzione di programma aperto: tutti i termini ground, cioè, sono nomi di teoria potenziali. Le teorie core non possono definire abducibili, le teorie estensione possono avere *solo* abducibili. Inoltre, introdurremo un nuovo predicato predefinito e una dichiarazione.

In particolare, introduciamo il predicato `instanceOf/2`, che consente di correlare un nome estensione e un nome core in modo da stabilire la relazione

super tra le corrispondenti teorie. Il goal

```
instanceOf(instance, class)
```

in cui `instance` sia un nome estensione, e `class` un nome core, dichiara l'unità core `class` come la super unità dell'unità estensione `instance`. La valutazione di questo goal, quindi, aggiunge questo vincolo (la relazione super) allo store. Diremo che l'unità `instance` è un'*istanza* della classe `class`.

La dichiarazione `attribute/1`, invece, può comparire nella definizione di ogni unità core:

```
:- attribute ⟨Preds⟩.
```

Qui, *`Preds`* rappresenta una sequenza di simboli di predicato che costituiscono l'insieme degli *attributi* definiti dall'unità in cui la dichiarazione compare.

Lo scopo della dichiarazione è quello di consentire di associare a ogni unità estensione un insieme di predicati abducibili singoli. In particolare, se $[c_n,\ldots,c_1]$ è il contesto associato a c_n, e `o` è un'istanza di c_n, l'unione degli attributi delle unità del contesto associato a c_n costituisce per definizione l'insieme degli abducibili singoli di `o`.

Ciò consente di definire classi che fungono effettivamente da template per le loro istanze, come si può vedere nell'esempio seguente. Le istanze possono poi essere create e configurate dinamicamente, obbiettivo primario di tutto il presente lavoro.

L'unico punto cruciale per comprendere l'evoluzione delle computazioni del linguaggio-campione è stabilire cosa avviene quando il contesto di binding si trova a essere un *contesto estensione*, ossia un contesto che non contiene solo unità core. In tutti gli altri casi, infatti, la computazione può seguire le regole introdotte nel capitolo 3.

Notiamo innanzitutto come, tramite le dichiarazioni `attribute/1`, il predicato `instanceOf/1` determini la conoscenza completa dell'insieme dei

predicati definiti da un'unità estensione, e, di conseguenza, dell'insieme di binding dei contesti associati a ogni istanza, anche se gli assiomi dell'unità istanza stessa non sono noti. Pertanto, il binding in un contesto estensione si sospende laddove non sia nota la struttura di tutte le unità estensione del contesto di binding. Non appena lo store contiene tutte le definizioni degli abducibili per tutte le teorie estensione coinvolte, risulta noto l'insieme di binding relativo al contesto di binding, ed è quindi possibile applicare direttamente la regola (4) del capitolo 3, che regola il binding non locale.

A questo punto, vi sono due possibilità: la prima è che la computazione prosegua localmente a un'unità core, e ciò non ha bisogno di essere descritto ulteriormente; la seconda è che la risoluzione del binding non locale porti a cercare di dimostrare una formula atomica $p(\tilde{t})$ in un'unità estensione o, che avrà ovviamente p tra i suoi singoli abducibili. Ciò risulta molto semplicemente nell'applicazione del vincolo di singolarità: il metavincolo di dimostrabilità $o{:}p(\tilde{t})$ viene aggiunto allo store globale, e la sua consistenza rispetto allo store viene verificata imponendo che, in presenza di un ulteriore metavincolo $o{:}p(\tilde{t}')$, sia $\tilde{t}=\tilde{t}'$.

Se poi, al termine di una computazione di successo, l'insieme dei vincoli $o{:}p(\tilde{t})$ avrà portato $\tilde{t}$ a non avere più variabili libere, il sistema potrà allora *abdurre* l'assioma $p(\tilde{t})$ relativamente alla teoria o, dando luogo così effettivamente a un passo di *configurazione* di o. Viceversa, dovrà limitarsi a esprimere in termini di metavincoli sull'unità o i risultati raggiunti.

Per una più precisa caratterizzazione operazionale di questo linguaggio d'esempio (ovvero, di una sua variante) si rimanda ancora a [OmNa94]. Si ritiene in questa sede più significativo riprendere l'esempio del capitolo 2, e mostrare come tutti i problemi presentati dai vari approcci trovino in questo approccio una soddisfacente soluzione.

5.2.2 Un esempio di programma

Riprendiamo l'esempio del resistore che ci ha accompagnato per tutto il

capitolo 2. La classe dei resistori potrà ora essere scritta per esempio nel modo seguente:

```
:- unit resistor.

:- attribute state/2, resistance/1.

ohm(V, R, I) :-                                          %a1
    xelf state(V,I), xelf resistance(R), V = R * I.

voltage(V) :- xelf state(V,_).                           %a2

value(R) :- xelf resistance(R).                          %a3

current(I) :- xelf state(_,I).                           %a4

equivalent(series(Res1,Res2)) :-                         %a5
    ohm(V,_,I), V = V1 + V2,
    Res1 <- ohm(V1,_,I), Res2 <- ohm(V2,_,I).

equivalent(parallel(Res1,Res2)) :-                       %a6
    ohm(V,_,I), I = I1 + I2,
    Res1 <- ohm(V,_,I1), Res2 <- ohm(V,_,I2).
```

In questo modo, qualunque unità estensione che ne sia dichiarata istanza avrà per definizione come abducibili singoli `state/2` e `resistence/1`, che, come già fatto osservare nel capitolo 4, fungeranno da attributi dell'istanza.

Dichiarando un'altra unità core come

```
:- unit colouredResistor isa resistor
:- attribute colour/1.
```

possiamo ora creare dinamicamente istanze di `colouredResistor` che avranno come attributi tanto `colour/1` (dichiarato in `colouredResistor`) quanto `state/2` e `resistance/1` (dichiarati in `resistor`), in quanto il contesto associato a `colouredResistor` comprende anche `resistor`.

Date queste dichiarazioni, quindi, l'effetto della valutazione del goal

```
instanceOf(r1,resistor)
```

o, equivalentemente,

```
r1 instanceOf resistor
```

è quello di aggiungere due nuovi abducibili {`r1:state/2`,

`r1:resistance/1}` all'insieme degli abducibili del programma e i corrispondenti vincoli di singolarità all'insieme dei vincoli d'integrità, accrescendo in tal modo lo store della computazione.

Analogamente,

```
r2 instanceOf colouredResistor
```

aggiunge tre nuovi abducibili singoli {`r2:state/2`, `r2:resistance/1`, `r2:colour/1`} allo store.

Vediamo ora come si sviluppa una computazione indotta dall'invocazione di un goal contestuale come il seguente

```
?- r1 instanceOf resistor, r1 <- current(2),          %g1
   r1 <- value(5), r1 <- ohm(V,_,_).
```

scomponendola in passi successivi ottenuti partendo dal suo primo sottogoal, invocato da solo, e aggiungendo ordinatamente uno alla volta gli altri sottogoal.

Nel primo passo

```
?- r1 instanceOf resistor.                            %g1′
r1 instanceOf resistor ?
```

`r1` viene dichiarata unità estensione, istanza di `resistor`, e quindi, come visto sopra, vengono dichiarati i relativi predicati abducibili singoli {`r1:state/2`, `r1:resistance/1`}. Si noti come il sistema, non avendo ancora potuto configurare `r1`, si limita a dare in uscita l'unico vincolo noto sull'unità, in maniera del tutto banale.

Il secondo sottogoal

```
?-  r1 instanceOf resistor, r1 <- current(2).        %g1″
r1 instanceOf resistor, r1:state(_,2) ?
```

si traduce nella richiesta di dimostrazione della formula `current(2)` rispetto al contesto associato a `r1`, ossia `[r1,resistor]`, chiamata che viene legata alla definizione di `resistor`. Questa definizione risulta poi nella chiamata `xelf state(_,2)`, che, risolta questa volta nell'unità `r1` (ove `state/2` è un

abducibile singolo), si traduce nel vincolo `r1:state(_,2)`, che in pratica stabilisce il valore della corrente di `r1`. Tuttavia, questo non consente ancora l'abduzione di alcun assioma, giacché `state/2` non è ancora compiutamente vincolato: quindi, l'uscita viene espressa in termini di metavincolo `r1:state(_,2)`.

Il terzo sottogoal produce una computazione analoga al secondo. Tuttavia, l'aggiunta del metavincolo `r1:resistance(5)` ha un effetto diverso, in quanto, essendo ground, pone il sistema in grado di abdurre l'assioma relativo al predicato `resistance/1` per l'unità `r1`. Ciò è messo in rilievo nell'uscita del sistema tramite il connettivo `::`, in `r1::resistor(5)`.

```
?- r1 instanceOf resistor, r1 <- current(2),          %g1"'
   r1 <- value(5).
r1 instanceOf resistor,
r1:state(_,2),
r1::resistance(5) ?
```

Infine, l'ultimo sottogoal forza gli attributi di `r1` a seguire la legge di Ohm, così da vincolare a un valore anche la caduta di potenziale, e portando a compimento la configurazione dell'istanza. In questo modo, il processo di abduzione giunge a completa realizzazione.

```
?- r1 instanceOf resistor, r1 <- current(2),          %g1
   r1 <- value(5), r1 <- ohm(V,_,_).
V = 10,
r1 instanceOf resistor,
r1::state(10,2), r1::resistance(5) ?
```

L'evoluzione incrementale degli elementi coinvolti nella computazione indotta dal goal (`g1`) mostra tra le altre cose come questo modello consenta di trattare gli oggetti parzialmente specificati. In particolare, oggetti parzialmente specificati possono essere dati come risultato della valutazione di un goal.

L'approccio data-driven, tramite l'adozione di un modello a vincoli delle computazioni, si risolve in pratica nella scelta dell'*atomo eseguibile left-most*, ossia l'atomo del risolvente più a sinistra tra quelli per cui sono disponibili informazioni sufficienti per la valutazione. Di conseguenza, il goal (`g1b`)

produce essenzialmente gli stessi risultati del goal (`g1`).

```
?- R1 <- current(2), R1 instanceOf resistor,          %g1b
   r1 <- value(5), r1 <- ohm(V,_,_), R1 = r1.
R1 = r1, V = 10,
r1 instanceOf resistor,
r1::state(10,2), r1::resistance(5) ?
```

Il primo sottogoal viene sospeso, per essere risvegliato subito dopo la valutazione del secondo (che fornisce l'informazione necessaria alla valutazione del primo), traducendosi poi nel vincolo `R1:state(_,2)`. Anche il terzo e il quarto sottogoal vengono sospesi, fintantoché la valutazione dell'ultimo sottogoal non completa l'informazione su `r1`, rendendo eseguibili i goal sospesi.

Il goal (`g2`) può suggerire una delle intuizioni fondamentali che sta dietro la nozione di creazione e configurazione dichiarativa di teorie logiche: da un certo punto di vista, un'unità estensione si comporta esattamente alla stessa stregue di un'unità core a essa identica. Grossolanamente, possiamo guardare le unità estensione come unità che *già esistono* (nell'interpretazione del programma), pur non comparendo nella descrizione testuale del programma, e che devono essere "rivelate" dalla computazione logica.

Infatti, un goal che, dato un programma *P*, porti a completa configurazione una teoria estensione *T*, produce gli stessi risultati (a parte ovviamente la configurazione di *T*) se invocato su un programma *P'* che contenga oltre alle unità di *P* un'ulteriore unità core identica a *T* configurata. Su questa semplice idea, in definitiva, si fonda la possibilità di creazione e configurazione dichiarativa di teorie logiche.

```
?- r1 instanceOf resistor, r2 instanceOf resistor,    %g2
   r2 <- equivalent(series(r1,r1)),
   r1 <- ohm(V,5,I), r2 <- current(2).
I = 2, V = 10,
r1 instanceOf resistor, r2 instanceOf resistor,
r1::state(10,2), r1::resistance(5),
r2::state(20,2), r2::resistance(10), ?
```

Infatti, la prova di (`g2`) avrebbe (sostanzialmente) lo stesso risultato (la

creazione e la configurazione dell'unità `r2`) anche se `r1` fosse un'unità core, dichiarata staticamente nel testo del programma (naturalmente, secondo il risultato di (`g2`)). Di più, se entrambe `r1` ed `r2` fossero state unità core, il goal (`g2`) avrebbe avuto analogo successo, legando banalmente `V` a `10` e `I` a `2`.

Si può così vedere come le stesse clausole usate per la configurazione di un'unità estensione possano essere ugualmente sfruttate per dimostrazioni logiche in unità completamente configurate, siano esse definite staticamente o dinamicamente. Di conseguenza, i programmi di questo framework possono essere scritti avendo in mente il solito modello della programmazione logica di computazione come deduzione, dato che la configurazione delle istanze è effettuata automaticamente dal sistema estraendo informazioni (via abduzione) dalle computazioni logiche.

Infine, la computazione indotta dalla valutazione del goal (`g2`) è particolarmente adatta a essere letta come l'evoluzione di una piccola rete di elementi computazionali indipendenti. Dopo la definizione di `r1` ed `r2` come istanze di `resistor`, il terzo sottogoal correla gli stati dei due resistori secondo la nozione di equivalenza tra resistori in serie. Da qui in avanti, ogni successiva evoluzione di uno dei due oggetti avrà effetti sull'altro: gli stati dei due oggetti sono quindi *vincolati* tra loro. Si noti, in particolare, che il quarto e quinto sottogoal, presi isolatamente, avrebbero configurato solo parzialmente `r1` e `r2`, rispettivamente. Siccome invece si trovano a essere valutati in un sistema globalmente vincolato, essi producono la configurazione completa di entrambe le istanze.

Si noti, inoltre, come la consistenza non sia più, per la stessa ragione, una questione legata allo stato di un singolo oggetto, ma piuttosto allo stato del sistema nel suo complesso. Per esempio, il tentativo di vincolare al valore `3` la corrente di `r1` con un ulteriore sottogoal, benché non contrasti con i vincoli direttamente espressi su `r1` stesso, fallirebbe a causa dei vincoli che correlano il sistema nel suo complesso (nel caso, gli stati di `r1` ed `r2` insieme).

5.3 Sommario

Tramite l'idea di (meta)vincolo di dimostrabilità, è possibile definire un modello computazionale in grado di realizzare il modello concettuale per la creazione e configurazione dichiarativa di teorie logiche basato sull'abduzione.

In tale vincolo, sia il contesto di prova sia la formula atomica da dimostrare sono elementi di una relazione di metalivello, che vincola il contesto a implicare logicamente la formula. Di conseguenza, sia la formula, sia (soprattutto) il contesto possono concettualmente essere configurati come risultato di una dimostrazione logica.

Un'intepretazione a vincoli consente di esprimere nella maniera più consona la caratterizzazione operazionale del modello, così come permette di definire semplici linguaggi che realizzino il modello stesso.

In particolare, il modello contestuale esteso introdotto nel capitolo 3 può essere facilmente esteso a comprendere la nozione di estensione di programma. Ciò consente al modello di catturare ed esprimere concetti come la creazione e configurazione dinamica di oggetti, e la relazione classe/istanza, realizzando in questo modo un modello di programmazione logica che può a pieno titolo dirsi orientato agli oggetti.

6 Conclusioni

6.1 Risultati

Il risultato fondamentale di questa tesi è di aver mostrato concretamente come i limiti posti finora all'integrazione del modello a oggetti nella programmazione logica possano essere superati.

Tutte le principali caratteristiche che rendono attraente l'approccio orientato agli oggetti possono essere riprodotti in ambito logico, senza rinunciare alle tipiche proprietà dei linguaggi logici, o complicare inutilmente il lavoro del programmatore logico. Il risultato è un paradigma di programmazione logica orientata agli oggetti che può essere istanziato in forma di linguaggi diversi.

Inoltre, la realizzabilità del modello definito è stata ampiamente dimostrata con un'implementazione completa e funzionante, e con una serie di verifiche in ambiti applicativi avanzati come la robotica.

6.2 Sviluppi futuri

Se molto è stato fatto in questi tre anni, a mio avviso, molto rimane da fare.

In primo luogo, lo sforzo di estensione non solo prototipale di CSM verso la configurazione di teorie logiche dovrà essere portato a termine in tempi brevi, e ne dovrà essere data una valutazione in termini di efficienza computazionale. Inoltre, la particolare estensione scelta dovrà essere caratterizzata semanticamente in modo completo e dettagliato, sia in senso operazionale sia in senso dichiarativo.

Rimangono poi aperte due questioni fondamentali: la relazione con il modello CLP(X), e la modifica dello stato.

Il modello a vincoli di metalivello dovrà pertanto essere comparato e sistemato dal punto di vista teorico rispetto al modello CLP(X) di [JaLa87], per definire punti di contatto e di differenziazione in maniera chiara.

Infine, si dovrà affrontare la questione della modifica di stato di un oggetto rappresentato da una teoria logica: se possibili approcci sono stati accennati al termine del capitolo 2, pure non è ancora stata individuata una soluzione soddisfacente e definitiva per questo problema, che diviene comunque il più urgente, e sarà quindi l'oggetto principale della ricerca futura.

A Cenni di logica del prim'ordine

A.1 Linguaggi e teorie del prim'ordine

A.1.1 Sintassi

Un *linguaggio logico del prim'ordine* è l'insieme delle formule ben formate che possono essere costruite a partire da un dato alfabeto, che è l'alfabeto della *teoria logica del prim'ordine* cui il linguaggio appartiene. Un siffatto alfabeto si compone di *costanti*, *variabili*, *funzioni*, *predicati*, *connettivi*, *quantificatori* e *simboli d'interpunzione*. (Si veda [Llo87] per un approfondimento).

Un *termine* di un linguaggio logico è una costante, una variabile o una funzione *n*-aria applicata a *n* termini. Un *atomo*, o *formula atomica*, è un predicato *n*-ario applicato a *n* termini. Una *formula ben formata* si costruisce combinando, secondo opportune regole, formule atomiche con connettivi (come $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), quantificatori ($\exists$, $\forall$) e simboli d'interpunzione (parentesi). Formalmente, se $\langle F \rangle$ denota una formula ben formata, $\langle A \rangle$ una formula atomica, e $\langle V \rangle$ una variabile del linguaggio, la seguente definizione denota l'insieme delle formule ben formate di un linguaggio logico.

$$\langle F \rangle ::= \langle A \rangle \mid (\langle F \rangle) \mid \neg\langle F \rangle \mid \langle F \rangle \wedge \langle F \rangle \mid \langle F \rangle \vee \langle F \rangle \mid \langle F \rangle \rightarrow \langle F \rangle \mid \langle F \rangle \leftrightarrow \langle F \rangle \mid \exists \langle V \rangle \langle F \rangle \mid \forall \langle V \rangle \langle F \rangle \qquad \text{(A.1)}$$

Le parentesi consentono di determinare l'ambito di connettivi e quantificatori, in assenza di regole di precedenza applicabili. Quella che segue è la tipica gerarchia di precedenza tra gli operatori, con la precedenza più alta propria degli operatori posti più in alto:

$$\begin{array}{c} \exists \ \forall \ \neg \\ \vee \\ \wedge \\ \rightarrow \ \leftrightarrow \end{array}$$

L'*ambito* di un quantificatore è la formula ben formata che segue una variabile quantificata. Ogni occorrenza di una variabile che non cada nell'ambito di un quantificatore è un'*occorrenza libera* della variabile stessa, altrimento, è un'*occorrenza legata*, o *bound*. Una formula si dice *chiusa* se tutte le occorrenze delle sue variabili sono legate, mentre si dice *ground* se non contiene variabili.

Dato un insieme di formule ben formate P, diciamo *Universo di Herbrand* l'insieme H_P di tutti i termini ground che si possono costruire combinando costanti e simboli di funzione che compaiono nelle formule di P. Diciamo poi *Base di Herbrand* l'insieme delle formule B_P ottenibili applicando i simboli di predicato che compaiono in P ai termini ground di H_P.

Un *letterale* è un atomo o la sua negazione. Una *clausola* è una particolare formula chiusa: una disgiunzione di letterali con tutte le variabili quantificate universalmente. Una generica clausola ha la forma

$$\forall \tilde{x} (A_1 \vee A_2 \vee \ldots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m) \qquad \text{(A.2)}$$

ove gli A_i e i B_j sono atomi, e $\tilde{x}$ è la tupla di tutte le variabili che compaiono nella clausola. Il tipico modo di scrivere una clausola deriva dalla seguente equivalenza:

$$\forall \tilde{x} (A_1 \vee A_2 \vee \ldots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m) \equiv$$
$$\forall \tilde{x} ((A_1 \vee A_2 \vee \ldots \vee A_n) \vee \neg (B_1 \wedge B_2 \wedge \ldots \wedge B_m)) \equiv$$
$$\forall \tilde{x} ((A_1 \vee A_2 \vee \ldots \vee A_n) \leftarrow (B_1 \wedge B_2 \wedge \ldots \wedge B_m))$$

L'ultima versione viene usualmente scritta sottintendendo quantificatori e parentesi, e sostituendo virgole ai connettivi $\wedge$ e $\vee$, cosicché le clausole sono usualmente rappresentate con la notazione

$$A_1, A_2, \ldots, A_n \leftarrow B_1, B_2, \ldots, B_m \qquad \text{(A.3)}$$

equivalente alla (A.2).

Quando $n = 1$,

$$A \leftarrow B_1, B_2, \ldots, B_m \qquad \text{(A.4)}$$

è una *clausola di programma*, o *clausola definita*, ossia una disgiunzione di letterali con un solo atomo positivo. Se in aggiunta vale anche $m = 0$,

$$A \leftarrow \qquad \text{(A.5)}$$

è una *clausola unitaria* (un solo atomo, positivo), o *fatto*. D'altra parte, una clausola con solo atomi negativi è un *goal definito*:

$$\leftarrow B_1, B_2, \ldots, B_m \qquad \text{(A.6)}$$

La clausola vuota $\square$ ha $m = n = 0$, e dev'essere letta come la *contraddizione logica* (falso o non vero).

Il letterale positivo A di una clausola definita è detto *testa* della clausola, mentre la congiunzione dei letterali negativi $B_1, B_2, \ldots, B_m$ rappresenta il *corpo* della clausola.

Una *clausola di Horn* è una clausola definita o un goal definito. Un *programma definito* è una collezione di clausole definite.

A.1.2 Interpretazione di un linguaggio logico del prim'ordine

Una volta data la struttura sintattica di un linguaggio logico del prim'ordine, ciò che resta da fare è fornirne la caratterizzazione semantica. La nozione di *interpretazione* di un linguaggio logico funge da ponte tra l'aspetto sintattico e quello semantico della programmazione logica, fornendo la connessione tra i simboli di un programma scritto in un linguaggio logico del prim'ordine e le astrazioni del dominio inteso.

Denotiamo con D il dominio del discorso, i cui elementi debbano essere rappresentati in un linguaggio logico. Connettivi, quantificatori e simboli d'interpunzione hanno un significato prefissato, mentre le variabili rappresentano generici elementi dell'Universo di Herbrand del linguaggio. Ciò che deve quindi essere usato per denotare gli oggetti di D sono costanti, funzioni e predicati.

Definiamo allora la nozione di *pre-interpretazione* di un linguaggio L, che consiste in:

1) il *dominio* dell'interpretazione, ossia un insieme D non vuoto;
2) un elemento di D per ogni costante di L;
3) una funzione da D^n a D per ogni simbolo di funzione n-ario di L.

Quindi, una pre-interpretazione consiste in effetti nell'associazione di un'elemento del dominio D a ogni elemento di H_L. D'altro canto, ogni elemento di H_L può essere visto come la rappresentazione di un oggetto del dominio del discorso secondo una data pre-interpretazione

Interpretando i simboli di predicato di un linguaggio logico, siamo in grado di introdurre il concetto di *verità* di una formula. Un'*interpretazione* consiste in una pre-interpretazione più

4) una relazione su D^n (oppure, equivalentemente, una funzione booleana su D^n) per ogni simbolo di predicato n-ario di L.

Un'interpretazione consente quindi di assegnare un valore di verità a ogni formula atomica. Combinando le regole seguenti, a ogni formula chiusa può essere assegnato un valore di verità secondo un'interpretazione I (basata a sua volta su una pre-interpretazione J). Una formula logica F è vera rispetto a un'intepretazione I data se

- F è atomica e ground, e la n-pla dei valori assegnati agli argomenti di F da J appartiene alla relazione su D^n corrispondente secondo I al simbolo di predicato di F.
- F è una formula del tipo $\neg G$, $G \wedge G'$, $G \vee G'$, $G \rightarrow G'$, or $G \leftrightarrow G'$, il valore di verità di G e G' è dato, e le entrate corrispondenti nella tabella di verità del connettivo portano a *true*.
- F è quantificata esistenzialmente, ed esiste un elemento di D (il dominio di J) che rende vera F rispetto a I. Cioè, F ha la forma $\exists x\, G$, e $\exists d \in D$ tale che la formula che si ottiene assegnando d a x in G è vera rispetto a I.[17]
- F è quantificata universalmente, e ogni elemento di D rende vera F rispetto a I. Cioè, F è del tipo $\forall x\, G$, e $\forall d \in D$ la formula che si ottiene assegnando d a x in G è vera rispetto a I.

Queste regole consentono quindi di assegnare a ogni formula chiusa un valore

---

[17] La nozione di *assegnamento di variabili* sarebbe qui necessaria, ma è lasciata all'intuizione del lettore, per non appesantire l'esposizione.

di verità, data un'interpretazione.[18]

Sul concetto di valore di verità di una formula si fonda poi la nozione di *modello*. Data una formula chiusa F di un linguaggio L del prim'ordine, un modello M per F è un'interpretazione M di L che rende vera F. Se S è un insieme di formule chiuse, allora M è un modello per S se M è un modello per ogni formula di S. In particolare, S è *soddisfacibile* se ammette (almeno) un modello; è *valido* se qualunque interpretazione di L è un modello per S.

Possiamo così definire il concetto fondamentale di *conseguenza logica*: se A e B sono formule chiuse di L, diremo allora che B è una conseguenza logica di A se ogni modello per A è un modello per B, e scriveremo $A \models B$. Se $A \models B$ e $B \models A$, allora A e B sono *logicamente equivalenti*. In particolare, se S è un insieme di formule chiuse, diremo che S *implica* la formula chiusa F ($S \models F$) qualunque modello di S è un modello per F. Una conseguenza fondamentale di queste definizioni è che

$$S \models F \quad \Leftrightarrow \quad S \cup \{\neg F\} \text{ non è soddisfacibile} \tag{A.7}$$

Dato un insieme di formule ben formate S, possiamo quindi denotare S tramite l'insieme delle sue conseguenze logiche, ossia l'insieme degli atomi ground che sono implicati da S. La denotazione *model-theoretic* di S è quindi

$$Decl(S) ::= \{A \in B_S \mid S \models A\} \tag{A.8}$$

Un'interpretazione che risulta di particolare utilità è la cosiddetta *interpretazione di Herbrand*. Scegliamo H_L come il dominio D della pre-interpretazione di Herbrand, associando ogni costante di H_L a se stessa, e ogni simbolo di funzione f n-ario alla funzione che mappa qualunque n-pla $a_1, \ldots, a_n \in H_L$ in $f(a_1, \ldots, a_n)$. A tale pre-interpretazione (che è *fissata* per tutte le interpretazioni di Herbrand di un linguaggio, e quindi non

---

[18] Se una formula non è chiusa, non è possibile assegnarle un valore di verità senza un assegnamento di variabili V, rendendo il valore di verità risultante dipendente da I e V, invece che da I sola.

caratterizzante) aggiungiamo la scelta di un sottoinsieme di B_L per ogni simbolo di predicato, così da completare la definizione dell'interpretazione di Herbrand.

Ciò che caratterizza un'interpretazione di Herbrand è quindi sostanzialmente un sottoinsieme della base di Herbrand, l'insieme $I \subseteq B_L$ delle formule atomiche che l'interpretazione rende vere; quindi, l'insieme delle possibili interpretazioni è l'insieme potenza di B_L.

Rispetto a un'interpretazione di questo tipo, è immediato specializzare la relazione di verità per clausole, nonché la nozione di modello. In particolare, dato un'insieme di formule ben formate *S*, diciamo *modello di Herbrand* di *S* un'interpretazione di Herbrand che sia anche un modello per *S*.

Nel caso delle clausole, i modelli di Herbrand godono di proprietà che risultano particolarmente utili. In primo luogo, l'esistenza di un modello per un insieme di clausole implica l'esistenza di un modello di Herbrand; in secondo luogo, l'assenza di un qualunque modello per un insieme di clausole equivale all'assenza di un qualunque modello di Herbrand. Inoltre, si vede facilmente che l'intersezione di modelli di Herbrand per insiemi di clausole definite è ancora un modello.

Dato che un modello di Herbrand esiste sempre (la Base di Herbrand), esiste sempre ed è unico il *modello minimo di Herbrand*, ottenuto come l'intersezione di tutti i modelli di Herbrand. Dato quindi un insieme di clausole definite S, esiste sempre ed è unico il modello minimo di Herbrand $M_S \subseteq B_S$.

La proprietà fondamentale del modello minimo di Herbrand è quello di coincidere con la denotazione model-theoretic: dato infatti un insieme di clausole definite *P*, si dimostra che

$$Decl(P) = M_P \qquad \text{(A.9)}$$

A.1.3 Metodo assiomatico

La nozione di conseguenza logica risulta impratica, poiché non è facile determinare il valore di verità di una formula in qualunque dominio non banale. In particolare, ciò che serve è un metodo costruttivo che possa essere utilizzato per un linguaggio di programmazione logico. Una *teoria assiomatica*, o teoria logica del prim'ordine, è costituita da una sintassi, un insieme di *assiomi*, e un insieme di *regole d'inferenza*. Mentre la sintassi è rappresentata da un linguaggio logico del prim'ordine, le regole di inferenza possono essere sfruttate per determinare il valore di verità di una formula del linguaggio a partire dagli assiomi (quelli *logici* e quelli *propri*).

L'idea fondamentale sottesa dal metodo assiomatico è quella di fornire uno strumento per il calcolo del valore di verità di una formula. Diciamo infatti che una formula F è *derivabile* in una teoria logica del prim'ordine T ($T \vdash F$) se in T è possibile costruire una dimostrazione per il teorema F . Una *dimostrazione* in T è una sequenza $F_1, \ldots, F_n$ di formule ben formate di T, in cui, per ogni i, F_i o è un assioma di T, o si ottiene da F_{i-1} applicando una regola d'inferenza. L'ultima formula della sequenza costituisce il *teorema* della dimostrazione. Così, $T \vdash F$ se esiste una dimostrazione di T il cui teorema è F.

Il metodo assiomatico per la determinazione del valore di verità di una formula è in effetti un metodo sintattico: le regole di inferenza effettuano una trasformazione puramente sintattica delle formule coinvolte. Rimane ovviamente da verificare, a questo punto, l'applicabilità di questo metodo, nonché la relazione tra la nozione di implicazione logica e quella di derivabilità di una formula.

Il primo risultato noto è che la determinazione del valore di verità di una formula in una teoria logica del prim'ordine è un problema semi-decidibile. In particolare, qualunque formula vera può essere derivata da un teoria logica del prim'ordine in un tempo finito, mentre non esiste un algoritmo che consenta di stabilire che una data formula non è un teorema.

Il secondo risultato è che il calcolo dei predicati del prim'ordine (con le regole d'inferenza di (*i*) scambio definizionale, (*ii*) modus ponens, e (*iii*) generalizzazione) è *corretto* e *completo*. Dati cioè un qualunque insieme di assiomi che caratterizzino una teoria logica del prim'ordine T, e una sua formula ben formata F, si dimostra che $T \models F \Leftrightarrow T \vdash F$: ogni formula derivabile da T è una sua conseguenza logica, e viceversa. Il passaggio a una procedura *automatica* di derivazione di teoremi da un insieme di assiomi logici è ora il tassello mancante verso la costruzione di un linguaggio logico di programmazione.

Il metodo più diffuso per la derivazione automatica di teoremi è il cosiddetto *Principio di Risoluzione* [Rob65], basato su un sistema formale privo di assiomi logici e costituito da un'unica regola d'inferenza, la *risoluzione*, che si applica a formule in forma clausale.

Questo metodo, assai più semplice ed efficiente del metodo assiomatico presentato in precedenza, non sarà qui dettagliato (per questo, si veda [CLM91]): basti sapere che su una serie di semplificazioni e una particolare strategia di applicazione (SLD, *Linear resolution with Selection rule for Definite Clauses*) del principio di risoluzione si basa la classica procedura di derivazione, particolarmente efficiente, adottata dai linguaggi di programmazione logica come Prolog.

Il risultato fondamentale, comunque, è che anche per la risoluzione SLD valgono le proprietà di correttezza e completezza. In particolare, se diciamo che $A \in B_P$ appartiene all'*insieme di successo* di P SS_P (ove P sia un insieme di clausole definite) se esiste una *refutazione* SLD per $P \cup \{\leftarrow A\}$, allora sussiste l'uguaglianza seguente

$$SS_P = Decl(P) = M_P \qquad \text{(A.10)}$$

Da tale risultato, infatti, deriva sostanzialmente la possibilità di definire linguaggi di programmazione logici dotati di caratterizzazione semantica dichiarativa, equivalente a quella operazionale.

B Teorie logiche etichettate

In questa appendice ci occuperemo di mostrare alcune implicazioni sintattiche e semantiche legate all'estensione dei linguaggi logici del prim'ordine con la nozione di *etichetta*, così come ricorrentemente riportato lungo tutto il presente lavoro. L'ambito scelto, per semplificazione essenzialmente formale, è quello dei linguaggi logici multiteoria, dove le teorie hanno nomi che sono elementi dell'Universo di Herbrand, ma non possono essere tra loro composte, e dove non esistono categorie di visibilità. Tutte le considerazioni seguenti possono poi essere estese senza grosso sforzo concettuale ad ambiti più complessi, come quelli presentati nei capitoli precedenti (come per esempio il framework contestuale esteso presentato nel capitolo 3).

Ovviamente, tutte le nozioni introdotte nella precedente appendice sono date qui per scontate.

B.1 Logica del prim'ordine con etichette

B.1.1 Formule, clausole e teorie etichettate

L'idea fondamentale della programmazione a oggetti sta nel suo modello dei dati: gli oggetti del dominio applicativo sono mappati in strutture di programma separate. Corrispondentemente, l'idea fondamentale dietro la nozione di *teoria logica etichettata* è far corrispondere a ciascun oggetto del dominio inteso una teoria logica separata. Ogni oggetto coinvolto in una computazione è caratterizzato da un insieme di assiomi che ne descrivono le proprietà. Poiché ogni pre-interpretazione di un linguaggio logico fa corrispondere elementi del dominio del discorso con elementi dell'Universo di Herbrand, ciò equivale ad assegnare un insieme di assiomi a ogni termine ground. Di conseguenza, un programma logico con teorie logiche etichettate non consiste più di un unico insieme di assiomi per la dimostrazione delle formule, ma di diverse collezioni di assiomi, ciascuna denotata da un nome.

Quindi, le clausole di un framework multiteoria non descrivono più verità universali, ma solo ciò che è vero *rispetto* a una particolare entità del dominio del discorso. La nozione di verità di una formula è quindi cambiata: una formula atomica F non è più vera o falsa rispetto a un programma P inteso globalmente, ma è tale rispetto a una data teoria T, ove T è l'etichetta che denota un particolare insieme di assiomi del programma P.

Si può quindi pensare di introdurre la nozione di *formula atomica etichettata* nella forma

$$T : F \tag{B.1}$$

dove T è un termine (che spazia quindi sul dominio dell'Universo di Herbrand), detto *etichetta*, che denota una teoria, ed F è una formula atomica

standard (secondo (A.1)). Se *theory*(*T*) è la collezione di assiomi denotata da *T*, allora la formula $T : F$ è vera se *theory*(*T*) $\models F$ nel senso standard.

Quindi, una formula in questo framework può contenere sia formule atomiche che formule atomiche etichettate. In particolare, una clausola avrà la forma

$$A \leftarrow LB_1, LB_2, \ldots, LB_m \qquad \text{(B.2)}$$

in cui ogni LB_i è una formula atomica o una formula etichettata, e una teoria logica etichettata sarà una collezione di formule di questo tipo. Siccome poi le formule non etichettate sono interpretate come "da dimostrare rispetto alla teoria in cui occorrono", esse possono essere lette come formule implicitamente etichettate con il nome della teoria cui appartengono testualmente, e quindi eventualmente riscritte esplicitamente secondo questa interpretazione.

Possiamo quindi pensare di riformulare tutte le clausole di un programma logico multiteoria come un insieme di *clausole etichettate* della forma

$$O{:}A \leftarrow O_1{:}B_1, O_2{:}B_2, \ldots, O_m{:}B_m \qquad \text{(B.3)}$$

prefissando ogni atomo non etichettato (tra cui la testa della clausola) con l'identificatore della teoria *O*, e lasciando immutati gli altri.[19]

B.1.2 Interpretazione (di Herbrand) etichettata

È facile immaginare come riformulare tutta la logica del prim'ordine secondo l'idea delle formule etichettate. Solo la nozione di interpretazione necessita di una sostanziale (benché minima) riformulazione esplicita, pur basandosi su una accezione di pre-interpretazione del tutto standard.

---

[19] Si noti come, se ogni programma multiteoria è una collezione di clausole etichettate, il vincolo sintattico di scrivere il programma stesso come collezione di teorie distinte con nome si traduce qui nel requisito di groundness per ciò che riguarda l'etichetta della testa di ogni clausola, che corrisponde al nome della teoria di appartenenza.

In particolare, diciamo *interpretazione etichettata* di un linguaggio logico etichettato L una pre-interpretazione J (su cui l'interpretazione si fonda, si veda la sottosezione A.1.2) più

4) una relazione su $D \times D^n$ (o, equivalentemente, una funzione booleana con dominio $D \times D^n$) per ogni simbolo di predicato n-ario di L.

Intuitivamente, le relazioni su D^n sono definite rispetto a ogni elemento di D, rispetto al quale possono essere vere o false. Corrispondentemente, ogni formula formula atomica ground può essere vera o falsa rispetto a un particolare oggetto del dominio applicativo, rappresentato da un termine ground. Siamo quindi ora in grado di determinare il valore di verità di ogni formula atomica ground etichettata.

Siano l, t_1, …, t_n termini ground, e p un simbolo di predicato n-ario. Siano poi l', t_1', …, t_n' gli elementi del dominio del discorso corrispondenti a l, t_1, …, t_n secondo la pre-interpretazione J data. Infine, sia p' la funzione booleana assegnata a p dall'interpretazione etichettata I data (basata su J). Allora, il valore di verità della formula etichettata $l{:}p(t_1,\ldots,t_n)$ può essere determinata calcolando il valore di $p'(l',t_1',\ldots,t_n')$ secondo I.

Ovviamente, una *interpretazione di Herbrand etichettata* è una interpretazione etichettata basata su una pre-interpretazione di Herbrand. In particolare, possiamo mappare qualunque interpretazione etichettata di Herbrand su un sottoinsieme della *Base di Herbrand etichettata*, ossia l'insieme LB_L delle formule ground etichettate che possono essere costruite a partire dai simboli di predicato di L applicati agli elementi di H_L. Quindi, un'interpretazione di Herbrand etichettata è l'insieme delle formule atomiche ground etichettate che l'interpretazione definisce vere. Si potrà quindi usare anche nel caso esteso (con etichette) l'identificazione delle interpretazioni di Herbrand con sottoinsiemi della Base di Herbrand.

La nozione di *modello etichettato*, nonché quella di modello minimo di Herbrand, segue in maniera del tutto ovvia. Tuttavia, nel seguito, per ragioni

di semplicità, cercheremo una caratterizzazione semantica che riparta da una formulazione classica dei linguaggi del prim'ordine, semplicemente accresciuti della molteplicità di teorie e della possibilità di delegare esplicitamente la prova di formule tramite goal aperti della forma *O*:*G*.

B.2 Semantica dichiarativa

B.2.1 Modelli di Herbrand ammissibili

Sia *T* una teoria etichettata di un programma *P* (*T*∈*P*) in un ambito multiteoria senza composizione. Siano definiti in modo del tutto intuitivo gli insiemi B_T e H_P. Sia *Open*(*T*) l'insieme di tutti i goal aperti della forma *O*:*G* che occorrono in *ground*(*T*). I goal aperti possono essere trattati solo in via ipotetica, poiché nessun'assunzione può essere fatta per ciò che riguarda la loro verità, che dipende da altre teorie. Se ogni sottoinsieme di *Open*(*T*) è quindi un insieme di *ipotesi*, la caratterizzazione semantica di *T* può allora procedere sfruttando la nozione di *modello ammissibile* (come in [BLM92a, BLM92b]): l'idea è di definire un modello minimo di Herbrand della teoria per ogni possibile insieme di ipotesi, ossia uno per ogni elemento dell'insieme potenza di *Open*(*T*).

Definizione B.1. *Modello di Herbrand ammissibile per una teoria***.**

Sia $H \subseteq Open(T)$ un insieme di ipotesi per *T*, e sia *ground*(*T*)/*H* la teoria ottenuta da *ground*(*T*) cancellando da esso tutti i goal aperti appartenenti ad *H* e tutte le clausole che contengono goal aperti che ad *H* invece non appartengono.

Se denotiamo con $M_T(H) \subseteq B_T$ il modello minimo di Herbrand di *ground*(*T*)/*H*, diciamo allora che $M_T(H)$ è il *modello di Herbrand ammissibile* di *T* secondo le ipotesi *H*.

Quindi, ogni teoria *T* è caratterizzata da una collezione di modelli ammissibili,

il cui insieme denoteremo con *AHM*(*T*).

B.2.2 Modello di un programma multiteoria

Poiché tutte le teorie etichettate di un programma multiteoria sono ora caratterizzate mediante modelli che dipendono da ipotesi sui modelli delle altre teorie, ciò che resta da fare è definire un operatore che ci consenta, tramite ripetute applicazioni, di risolvere le interdipendenze tra teorie logiche, il cui risultato dovrà essere un unico modello (etichettato) per il programma. La definizione di tale operatore è molto simile a quella relativa a un operatore analogo riportata in [BLM92b].

Definizione B.2. *Operatore di composizione* ϕ_P**.**

Dato un programma *P* multiteoria con etichette, l'operatore di composizione ϕ_P è una funzione

$$\phi_P : 2^{LB_P} \rightarrow 2^{LB_P}$$

definita formalmente come segue:

$$\phi_P ::= \lambda I\ I \cup \{T{:}A \mid T \in P \wedge A \in M_T(H) \wedge H \subseteq I\}$$

in cui *I* è un insieme di formule atomiche ground etichettate.

È facile dimostrare che ϕ_P è *monotòno* e *continuo*. Per maggiori dettagli sull'apparato formale su cui si fondano definizioni e proprietà di questo capitolo, si consulti [Sch86].

Proposizione B.1. *Monotonicità di* ϕ_P**.**

È banale. Basta mostrare che

$$\forall I, I' \in 2^{LB_P}, I \subseteq I' \rightarrow \phi_P(I) \subseteq \phi_P(I').$$

Infatti, $\forall T{:}A \in LB_P$, si ha

$$
\begin{array}{cl}
T{:}A \in \phi_P(I) & \\
\updownarrow & (\textit{def. } \phi_P) \\
T{:}A \in I \vee (A \in M_T(H) \wedge H \subseteq I) & \\
\downarrow & (I \subseteq I') \\
T{:}A \in I' \vee (A \in M_T(H) \wedge H \subseteq I') & \\
\updownarrow & (\textit{def. } \phi_P) \\
T{:}A \in \phi_P(I') &
\end{array}
$$

Proposizione B.2. *Continuità di* ϕ_P**.**

Sia $X \subseteq 2^{LB_P}$ una catena. Allora l'operatore ϕ_P è continuo se $\phi_P(lub(X)) = lub(\{\phi_P(I) | I \in X\})$. Basta allora mostrare che, $\forall T{:}A \in LB_P$,

$$
\begin{array}{cl}
T{:}A \in \phi_P(lub(X)) & \\
\updownarrow & (\textit{def. } \phi_P) \\
T{:}A \in lub(X) \vee (A \in M_T(H) \wedge H \subseteq lub(X)) & \\
\updownarrow & (\textit{def. lub}) \\
\exists I \in X\ (T{:}A \in I \vee (A \in M_T(H) \wedge H \subseteq I)) & \\
\updownarrow & (\textit{def. } \phi_P) \\
\exists I \in X\ T{:}A \in \phi_P(I) & \\
\updownarrow & (\textit{def. lub}) \\
T{:}A \in lub(\{\phi_P(I) | I \in X\}) &
\end{array}
$$

Per questo, ϕ_P ha un minimo punto fisso $lfp(\phi_P) = \phi_P{\uparrow}\omega$. Possiamo quindi definire la denotazione di un programma P in un ambito logico con teorie multiple etichettate senza composizione nel modo che segue.

Definizione B.3. *Modello per un programma a teorie multiple etichettate***.**

Sia P un programma nel linguaggio logico con teorie multiple etichettate. Allora, il modello etichettato di P, che indichiamo con $||P||$, si ottiene applicando $\phi_P{\uparrow}\omega$ all'elemento minimo $\perp$ del reticolo completo 2^{LB_P}, che è ovviamente $\varnothing$. Definiamo quindi

$$||P|| ::= \phi_P \uparrow \omega(\varnothing)$$

Per le proprietà dell'operatore, tale modello esiste ed è unico.

È chiaro che $||P|| \subseteq LB_P$, per cui possiamo pensare di considerarlo come l'insieme di tanti modelli di Herbrand quante sono le teorie del programma *P*.

Bibliografia

[AiNa86] H. Aït-Kaci, R. Nasr. *LOGIN. A Logic Programming Language with Built-in Inheritance*. Journal of Logic programming, 3(3), 1986, pp. 185-215.

[AnPa90] A. Andreoli, R. Pareschi. *Linear Objects: logical processes with built-in inheritance*. Proceedings of ICLP'90, 1990.

[APPE] DS Logics. *APPEAL User's Manual*. Bologna, 1993.

[BGLM92] A. Bossi, M. Gabbrielli, G. Levi, M.C. Meo. *Contributions to the Semantics of Open Logic Programs*. Proceedings of the International Conference on Fifth Generation Computer Systems, 1992. ICOT 1992.

[BLM90] A. Brogi, E. Lamma, P. Mello. *A General Framework for Structuring Logic Programs*. Rapporto Tecnico C.N.R. "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", nº 4/1, Maggio 1990.

[BLM91] A. Brogi, E. Lamma, P. Mello. *Open Logic Theories*. L.-H. Eriksson, P. Krueger, P. Schroeder-Heister (ed.) Proceedings of Second Workshop on Extension of Logic Programming, LNAI, Springer-Verlag.

[BLM92a] A. Brogi, E. Lamma, P. Mello. *Objects in a Logic Programming Framework*. A. Voronkov (ed.), Logic Programming, LNAI, Vol. 592, pp. 102-113. Springer-Verlag, 1992.

[BLM92b] A. Brogi, E. Lamma, P. Mello. *Compositional Model-Theoretic Semantics*. New Generation Computing, 11, 1992. pp. 1-21.

[BLM94] M. Bugliesi, E. Lamma, P. Mello. *Modularity in Logic Programming*. Ten Years of Logic Programming, numero speciale del Journal of Logic Programming. Elsevier, New York (USA).

[BLMM91] A. Brogi, E. Lamma, P. Mancarella, P. Mello. *Abductive Reasoning in a Multi-theory Framework*. S. Gaglio (ed.), Trends in Artificial Intelligence, Proceedings of the Second Congress of IAAI, LNAI, Vol. 549, Springer-Verlag, 1991. pp. 137-146.

[Bolc94] D. Bolcioni. *Un linguaggio per la programmazione logica a oggetti: dalla teoria all'implementazione*. Tesi di Laurea, Università di Bologna, Ottobre 1994.

[Bor87] A. Borning, R. Duisburg, B. Freeman-Benson, A. Kramer, M. Wolf. *Constraint Hierarchies.* Proc. of OOPSLA '87, ACM 1987, pp. 48-60.

[Bug92] M. Bugliesi. *A declarative view of inheritance in logic programming*. K. Apt (ed.), Proceedings of the Joint International Conference and Symposium on Logic Programming, The MIT Press, 1992. pp. 113-130.

[CLM91] L. Console, E. Lamma, P. Mello. *Programmazione Logica e Prolog*. UTET, Torino (Italia), 1991.

[Con88] J. Conery. *Logical objects*. Proceedings of the Fifth International Conference and Symposium on Logic Programming. Seattle, 1988.

[CSM] E. Denti, A. Natali, A. Omicini. *CSM User's Guide*. Rapporto Tecnico Progetto Finalizzato CNR Sistemi Informatici e Calcolo Parallelo' nª 4/70, Roma, Maggio 1992.

[DLMNO93] E. Denti, E. Lamma, P. Mello, A. Natali, A. Omicini. *Techniques for Implementing Contexts in Logic Programming*. E. Lamma, P. Mello (ed.), Extensions of Logic Programming, LNAI nº 660, Springer-Verlag, Heidelberg, Germania, 1993, pp. 339-358.

[DNO92] E. Denti, A. Natali, A. Omicini. *Contexts as first-class objects: an implementation based on the SICStus Prolog system*. Atti del Settimo Convegno sulla Programmazione Logica, a cura di S. Costantini, 17-19 Giugno 1992, pp. 307-320.

[DNO93] E. Denti, A. Natali, A. Omicini. *Verso un ambiente di sviluppo per sistemi in tempo reale*. Rapporto Tecnico Progetto Finalizzato CNR Sistemi Informatici e Calcolo Parallelo' nª 4/91, Roma, Marzo 1993.

[DNO94] E. Denti, A. Natali, A. Omicini. *Moving Prolog Toward Objects*. E. Tick, G. Succi (eds.), Implementations of Logic Programming Systems. Kluwer, 1994. pp. 92-104.

[DNOZ94a] E. Denti, A. Natali, A. Omicini, F. Zanichelli. *A Structured Logic Programming Approach to Robot Programming*. Proceedings of the Second International Conference on Practical Applications of Prolog, PAP'94, Londra (UK), Aprile 1994.

[DNOZ94b] E. Denti, A. Natali, A. Omicini, F. Zanichelli. *Robot Control Systems as Contextual Logic Programs*. C. Beierle, L. Plümer (ed.), Logic Programming: Formal Methods and Practical Applications. Elsevier, 1994.

[Eclipse] *Eclipse User's Manual*. ECRC, 1991.

[FrBe90] B. Freeman-Benson. *Kaleidoscope: Mixing Objects, Constraints and Imperative Programming*. Proceedings of ECOOP/OOPSLA '90, 1990, pp. 77-87.

[JaLa87] J. Jaffar, J.L. Lassez. *Constraint Logic Programming*. Proc. 1987 ACM Symposium on Principles of Programming Languages, pp. 111-119, Monaco, Gennaio 1987.

[JaMa94] J. Jaffar, M.J. Mahrer. *Constraint Logic Programming: A survey*. Ten Years of Logic Programming, numero speciale del Journal of Logic Programming. Elsevier, New York (USA).

[KKT92] A. Kakas, R. Kowalski, F. Toni. *Abductive Logic Programming*. Journal of Logic and Computation, Vol. 2, 1992. pp. 719-770.

[KoSa87] R.A. Kowalski, F. Sadri. *An Application of General Purpose Theoerm-proving to Database Integrity*. Minker (ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, Palo Alto, 1987. pp. 313-362.

[Kow74] R.A. Kowalski. *Predicate Logic as a Programming Language*. Proc. IFIP '74, North Holland Publishing Co., Amsterdam (Olanda), pp. 569-574, 1974.

[Lam90] E. Lamma. *Programmazione Logica Strutturata: una Realizzazione in Ambiente Compilato*. Tesi di Dottorato, Università di Bologna, Febbraio 1990.

[Llo87] J.W. Lloyd. *Fondamenti di Programmazione Logica*. Franco Muzzio Editore, Italia, 1987. Edizione italiana di *Foundations of Logic Programming*. Springer-Verlag, 1987.

[LMNO91] E. Lamma, P. Mello, A. Natali, A. Omicini. *Due approcci per la realizzazione di un linguaggio logico strutturato in ambiente compilato*. Rapporto Tecnico Progetto Finalizzato CNR Sistemi Informatici e Calcolo Parallelo' nª 4/33, Roma, Luglio 1991.

[McC92] F.G. McCabe. *Logic and Objects*. Prentice Hall International, Londra, 1992.

[MeNa92] P. Mello, A. Natali. *Extending Prolog with Modularity, Concurrency and Metarules*. New Generation Computing, 10(4), Agosto 1992.

[Mil89] D. Miller. *A logical analysis of modules in logic programming*. Journal of Logic Programming, 6, pp. 79-108, 1989.

[MoPo89] L. Monteiro, A. Porto. *Contextual Logic Programming*. G. Levi, M. Martelli (ed.), Proceedings of the 6th International Conference on Logic Programming. The MIT Press, Cambridge (USA), 1989.

[NaOm93] A. Natali, A. Omicini. *Objects with State in Contextual Logic Programming.* M. Bruynooghe, J. Penjam (ed.), Programming Language Implementation and Logic Programming, LNCS, Vol. 714, pp. 220-234. Springer-Verlag, New York, 1993.

[NOZ93] A. Natali, A. Omicini, F. Zanichelli. *Exploiting Logic Programming in Robot Applications*. Atti dell'Ottavo Convegno sulla Programmazione Logica, a cura di D. Saccà, 15-18 Giugno 1993, pp. 535-548.

[ODN95] A. Omicini, E. Denti, A. Natali. *Multi-Agent Communication through Logic Theories*. Sottomesso per la pubblicazione.

[OmNa94] A. Omicini, A. Natali. *Object-Oriented Computations in Logic Programming*. M. Tokoro, R. Pareschi (ed.), Object-Oriented Programming. LNCS 821, Springer-Verlag, 1994. pp. 194-212.

[Pei58] C.S. Peirce. *Collected Papers of Charles Sanders Peirce*. Vol. 2, 1931-1958, Hartshorn *et al.* (ed.), Harvard University Press.

[Poo88] D. Poole. *A logical framework for default reasoning* In: Artificial Intelligence, 36, 1988. pp. 27-47.

[Rob65] J.A. Robinson. *A machine-oriented logic based on the Resolution Principle*. Journal of ACM, vol. 12, n° 1, 1965, pp. 23-41.

[Rog78] R. Rogers. *Logica matematica e teorie formalizzate*. Feltrinelli, Milano (Italia), 1978.

[Sch86] D.A. Schmidt. *Denotational Semantics*. Allyn & Bacon, Newton (Massachusetts), 1986.

[ShTa83] E. Shapiro, A. Takeuchi. *Object Oriented Programming in Concurrent Prolog*. New Generation Computing, 1(1), 1983.

[SICS] Swedish Institute of Computer Science. *SICStus User's Manual*. SICS, Kista, Sweden, 1994.

[Tre82] P.C. Treleaven, D.R. Brownbridge, R.P. Hopkins. *Data-Driven and Demand-driven Computer Architecture*. ACM Computing Surveys, Vol. 14, nº 1, marzo 1982.

[Ven95] M. Venuti. *Condivisione della conoscenza, comunicazione e inferenza nei sistemi cliente/servitore*. Tesi di Laurea, Università di Bologna, Febbraio 1995.

[Weg87] P. Wegner. *Dimensions of Object-based Language Design*. Proceedings of OOPSLA '87. ACM, 1987.

[Weg90] P. Wegner. *Concepts and Paradigms of Object-Oriented Programming.*. OOPS Messenger, Vol. 1, nº 1, Agosto 1990, ACM Press. pp. 7-87.

[Weg92a] P. Wegner. *Dimensions of Object-Oriented Modeling*. IEEE Computer, October 1992, pp. 12-20.

[Weg92b] P. Wegner. *Object-based Versus Logic Programming*. Proceedings of the International Conference on Fifth Generation Computer Systems, 1992. ICOT 1992.

[Zan84] C. Zaniolo. *Object Oriented Programming in Prolog*. Proceedings of the International Symposium on Logic Programming, Atlantic City, 1984.

[ZCNO94] F. Zanichelli, S. Caselli, A. Natali, A. Omicini. *A Multi-Agent Framework and Programming Environment fot Autonomous Robotics*. Proceedings of the International Conference on Robotics and Automation, ICRA '94, S. Diego, Maggio 1994.