Un' architettura riflessiva per i linguaggi logici

Stefania **Costantini** Gaetano Aurelio **Lanzarone**

Universita' degli Studi di Milano
Dipartimento di Scienze dell' Informazione
Via Moretto da Brescia 9, I-20133, Milano (+39-2-2772.222)

Sommario

Il presente lavoro si colloca nel contesto della discussione sui paradigmi evoluti basati sulle clausole di Horn. Viene proposta la realizzazione dei costrutti per definire e manipolare teorie e del predicato *demo(T, G)*, che dichiarativamente rappresenta l' asserzione metateorica della dimostrabilita' della proposizione *G* nella teoria *T*, sulla base di un' architettura logico-riflessiva basata su un unico meccanismo fondamentale, la *riflessione procedurale*. Tale architettura prevede la rappresentazione in clausole di Horn del predicato *demo*, di cui e' garantita la connessione causale con l' interprete reale, ed e' adatta alla realizzazione delle funzionalita' di un sistema logico orientato ad applicazioni di Intelligenza Artificiale. A questo proposito vengono presentate (discutendone la possibile realizzazione) e illustrate tramite esempi nuove versioni di *demo*, orientate al ragionamento ipotetico.

Introduzione

La convinzione della necessita' e opportunita' di rappresentare la conoscenza nei sistemi di deduzione su diversi livelli collegati fra loro e' da tempo largamente condivisa. Sono stati proposti diversi paradigmi per la rappresentazione ed uso di metaconoscenza, e di conseguenza si e' posto il problema di astrarre i concetti essenziali su cui deve basarsi un sistema che permetta di combinare rappresentazione ed uso di conoscenza e metaconoscenza ([Aie86], [Mae87]). Sono inoltre in corso tentativi di definire un contesto formale in cui sia possibile inquadrare i paradigmi esistenti [Gen87].

Un aspetto rilevante evidenziato in [Aie86] e' la necessita' di prevedere meccanismi uniformi e generali per la manipolazione della conoscenza piuttosto che soluzioni "ad hoc" per applicazioni specifiche. Su questa linea il presente lavoro,

Questo lavoro e' stato svolto presso l' Universita' degli Studi di Milano, Dipartimento di Scienze dell' Informazione, nell' ambito di uno studio congiunto con IBM Italia s.p.a.

che si colloca nel contesto della logica a clausole di Horn, prende spunto dal tentativo di [Ghi87] e [Ghi87a] di generalizzare ai linguaggi logici il modello di architettura riflessiva proposto in [Smi84], [DeR84]. L' obiettivo e' mostrare che tale modello, che prevede un unico meccanismo fondamentale, presenta sufficiente generalita' e flessibilita' per costituire una base per la realizzazione di paradigmi piu' complessi.

Ai § 1, 2 e 3 si introdurra' il modello e se ne discutera' l' applicazione alla programmazione logica partendo dal tentativo di [Ghi87]. Si esporranno inoltre le motivazioni che inducono ad affermarne la valenza unificante rispetto ad alcuni dei paradigmi finora proposti in quest' ambito.

Per esemplificare la possibilita' di implementare meccanismi complessi, al § 4 si discutera' la realizzazione del predicato *demo*, che rappresenta la dimostrabilita' nella logica a clausole di Horn, in modo alternativo sia alla definizione come metainterprete [Bow82] che alla realizzazione a basso livello ([Bow85], [Aie86]) in quanto questa definizione di *demo* e' *causalmente connessa* all' interprete reale. Al § 5 verranno presentati alcuni esempi dell' uso di *demo*, in margine ai quali si cogliera' l' occasione per aggiungere qualche osservazione in merito all' uso della metaconoscenza e alla potenza espressiva che ne risulta. Gli esempi verranno presentati utilizzando la sintassi Prolog nella versione di Edimburgo [Fur88].

Infine al § 6 si accennera' per grandi linee agli sviluppi previsti rispetto ai temi trattati nel presente lavoro.

1. Il modello e la sua realizzazione

L' architettura riflessiva si basa sulla considerazione per cui un processo computazionale puo' intendersi suddiviso in due componenti. La prima e' il cosiddetto *campo strutturale*, composto da strutture dati e programmi. La seconda e' l' interprete, o *processore*, che manipolando il campo strutturale da' luogo alla computazione. Il processore puo' essere primitivo, ossia realizzato mediante la macchina fisica, oppure puo' essere a sua volta decomposto in modo analogo.

Si supponga che il campo strutturale sia descritto in un linguaggio L di cui sia definito un *metaprocessore circolare* (MPC), ossia un interprete di L scritto nel linguaggio L stesso. Se il processore che manipola il campo strutturale si scompone in termini di una copia di MPC e di un secondo processore, e se anche questo si scompone allo stesso modo, e cosi' via all' infinito, si ottiene una torre alla cui base c'e' un programma P in linguaggio L (e i dati relativi), al primo livello una copia di MPC che interpreta P, ed a ciascuno dei livelli superiori c' e' una copia di MPC che interpreta un' altra copia di MPC. In sostanza, a tutti i livelli superiori al primo c' e' una istanza del metaprocessore circolare che interpreta se stesso. Si deve immaginare un processore primitivo che, alla sommita' della torre, esegua effettivamente una copia di MPC.

Se il metaprocessore MPC viene arricchito dalla descrizione di un atto riflessivo che consenta di spostare l' attenzione da un livello ad un altro, la torre diventa una *torre riflessiva*. Il meccanismo di connessione fra livelli proposto in [Smi84] in relazione all' ipotesi di un' architettura basata su un linguaggio funzionale e' la cosiddetta *riflessione procedurale*. Prevede un particolare tipo di procedura, la procedura riflessiva, che viene eseguita non al livello in cui e' invocata, ma al livello superiore, ed ha fra i suoi argomenti le strutture dati manipolate dal metaprocessore di quel livello (le quali rappresentano lo stato della computazione del livello dove la procedura e' stata chiamata), che essa puo' esaminare e/o modificare. Al termine dell' esecuzione della procedura riflessiva la computazione al livello della chiamata riprendera' da dove era stata sospesa se lo stato non ha subito modifiche, altrimenti si evolvera' in accordo alle modifiche stesse. La riflessione procedurale garantisce quindi la *connessione causale* fra livelli, requisito fondamentale per la significativita' del modello [Gen87].

L' architettura risultante e' un' architettura introspettiva [Smi86] [Mae87] in quanto un sistema in essa implementato dispone di un modello di se stesso, e conseguentemente e' in grado di analizzare e/o modificare il proprio comportamento. Il modello di se stesso e' composto dall' aspetto procedurale, essendo la descrizione procedurale del linguaggio a comporre la torre riflessiva, e dalla codifica dello stato attuale della computazione, esplicitata all' atto della riflessione. Al quesito su quando sia opportuno riflettere [Gen87], ossia decidere di spostare l' attenzione dalla soluzione di un problema al processo di soluzione, la risposta della riflessione procedurale e' che cio' sia determinato da un atto volontario del programma.

E' opportuno notare che, pur essendo gli infiniti livelli della torre in esecuzione simultaneamente, in ogni istante solo ad un determinato livello si svolge computazione utile. Normalmente il livello significativo e' il livello oggetto; la chiamata di una procedura riflessiva sposta pero' l' elaborazione utile al livello superiore, e chiamate innestate di procedure riflessive salgono di altrettanti livelli nella torre; la terminazione di una procedura riflessiva causa invece il ritorno al livello sottostante. E' quindi possibile implementare in modo finito la torre riflessiva tramite un unico processore reale il quale emuli in ogni istante il processore del livello su cui e' fissata l' attenzione e modelli i passaggi di livello, a condizione di poter ricostruire ad ogni passaggio lo stato del livello a cui ci si porta (escludendo ovviamente di mantenere un numero infinito di stati).

In [DeR84] viene mostrato, in relazione ai linguaggi funzionali, che se il metaprocessore e' tail–ricorsivo (in sostanza, se le funzioni del processore si

chiamano fra loro tail–ricorsivamente [All78] e senza chiamate innestate) ai livelli in cui MCP interpreta se stesso le sue strutture dati non subiscono modifiche. Ne consegue che ai livelli mai raggiunti lo stato e' uguale a quello iniziale, e in ciascuno degli altri e' proprio quello determinato dall' ultima procedura riflessiva eseguita a quel livello. In questo caso il comportamento della torre riflessiva si puo' quindi modellare tramite un unico processore (detto *level–shifting–processor*) il quale sia in grado di commutare fra i livelli, registrando lo stato di ciascun livello che venga abbandonato per poterlo ripristinare quando venga nuovamente raggiunto. Lo spostamento di un livello verso l' alto (*shift–up*) comporta l' attivazione del processore di livello superiore per elaborare l' espressione che ha causato lo shift (tipicamente la chiamata di una procedura riflessiva). Lo spostamento verso il basso (*shift–down*) avviene, concettualmente, quando il processore del livello corrente rileva di stare eseguendo se stesso, e quindi ritiene terminata l' elaborazione utile a quel livello; lo shift–down ha luogo in pratica quando il processore, elaborando una invocazione di funzione, si accorge che si tratta di una chiamata a determinate procedure facenti parte del processore stesso.

Qualsiasi linguaggio puo' essere scelto per comporre la torre riflessiva (anche se e' stata originariamente definita in relazione ad un dialetto del linguaggio LISP, il 3–LISP), a condizione che sia *semanticamente razionalizzato* [Smi84], nel senso che le relative strutture linguistiche possano suddividersi in categorie caratterizzate in modo univoco da notazione lessicale, contenuto dichiarativo e conseguenza procedurale. Le ragioni sono legate alla definizione del metaprocessore ed alla necessita' di estendere il linguaggio con capacita' metastrutturali, ossia la possibilita' di referenziare a ciascun livello espressioni linguistiche del livello inferiore (la presenza di un meccanismo di "naming" e' un requisito fondamentale per la formalizzazione del ragionamento su diversi livelli [Bow82], [Aie86], [Gen87]).

2. Applicazione alla programmazione logica: il tentativo del Logic

L' esperienza riportata in [Ghi87] e [Ghi87a] costituisce un primo passo nella direzione delle architetture riflessive basate sui linguaggi logici: con l' obiettivo della costruzione di un ambiente integrato logico–funzionale e' stato definito il Logic, un linguaggio basato sulle clausole di Horn e sulla riflessione procedurale, sviluppato in modo da risultare il piu' possibile analogo al 3–LISP (nell' ottica dell' integrazione). Gli aspetti rilevanti del Logic rispetto al tema generale della definizione di architetture logico–riflessive sono sintetizzati di seguito.

Un problema importante per la razionalizzazione semantica di un linguaggio basato sulle clausole di Horn riguarda il meccanismo di inferenza. Come infatti notato anche in [Bow85], gli usuali interpreti Prolog restituiscono all' utente una parte della prova (sotto forma dei valori di unificazione delle variabili che appaiono nella meta), cosa che coinvolge un passaggio al metalivello, dove sono visibili le strutture dati utilizzate dall' interprete ed in particolare le associazioni variabile/valore ottenute tramite le unificazioni. Per evitare quest' ambiguita' semantica, il Logic e' basato (analogamente al Prolog) sulla refutazione per risoluzione con strategia SLD [Llo84], ma il risultato di una refutazione e' sempre un valore di verita', *true* o *false* a seconda che la refutazione riesca o meno. I valori delle variabili (che l' utente considera come "risultato") possono essere ottenuti mediante riflessione.

La razionalizzazione richiede inoltre che il Logic sia un linguaggio logico puro, che non preveda predicati extra–logici di alcun genere (con l' eccezione dei meccanismi di referenziazione discussi piu' avanti). Il presupposto per l' utilizzo di un linguaggio in un' architettura riflessiva e' infatti che esso sia semanticamente ben fondato, e quindi privo di parti che non possano essere definite nel modello su cui il linguaggio e' basato. Per converso la riflessione rende possibile definire tutte le necessarie estensioni senza pregiudicare la base concettuale del linguaggio stesso. Per la definizione della sintassi e delle categorie semantiche del Logic si rimanda ai lavori citati.

Una caratteristica indispensabile per un linguaggio che debba essere usato per rappresentare conoscenza su piu' livelli e', come gia' accennato, la capacita' metastrutturale. A tale proposito il Logic prevede, analogamente al 3–LISP, la categoria degli "handle", che sono riferimenti alla rappresentazione degli oggetti del linguaggio. La notazione e' ottenuta premettendo il carattere "'" all' espressione linguistica riferita: ad esempio, il riferimento alla rappresentazione della formula a(1) si indichera' come 'a(1). Sono previsti i predicati primitivi "up" e "down" per referenziare e dereferenziare un oggetto. La presenza di tali predicati apparentemente pregiudica la purezza del linguaggio: si rimanda a [Gen87] per una discussione sui meccanismi di referenziazione, e una giustificazione della definizione procedurale di tali meccanismi.

Le strutture dati che in Logic si e' scelto di rendere esplicite all' atto della riflessione sono l' ambiente e l' albero di ricerca (o albero AND–OR, [Fur88]). L' ambiente contiene l' insieme delle definizioni dei predicati. La rappresentazione adottata e' analoga alle liste di associazioni del LISP [All78], dove ogni predicato e' rappresentato dalla coppia <nome, lista_clausole>. L' albero di ricerca rappresenta lo stato corrente della refutazione, inclusi i legami delle variabili: quando sia nota la strategia di risoluzione, l' esame e/o la manipolazione dell' albero permettono di visionare lo stato presente della refutazione e/o condizionarne l' evoluzione futura. La particolare rappresentazione adottata in Logic (per la quale si rimanda ai lavori citati) e' definita in modo da non essere legata a priori ad alcuna particolare strategia di risoluzione.

La scelta di utilizzare una rappresentazione dello stato notevolmente piu' complessa rispetto allo stack adottato come struttura base dagli interpreti Prolog risulta cosi' giustificata dalla possibilita' di influenzare la dimostrazione mediante elaborazione introspettiva.

Il meccanismo di riflessione in Logic non puo' essere realizzato in modo del tutto analogo al 3–LISP (procedure riflessive dichiarate staticamente come tali tramite una parola chiave) in quanto le uniche proprieta' statiche di un predicato sono il nome ed il numero degli argomenti. Di conseguenza e' necessario che un qualunque predicato possa, a richiesta, essere utilizzato in modo riflessivo: se alla chiamata si specifica un *atto riflessivo,* agli argomenti esplicitamente forniti ne verranno aggiunti altri due, legati appunto all' ambiente ed all' albero correnti. Inoltre la chiamata verra' eseguita (come per le procedure riflessive) al livello superiore, come se facesse parte del codice del processore.

Per specificare gli atti riflessivi e' stato aggiunto al linguaggio un nuovo tipo di oggetto, il *reflecter*, che specifica l' intenzione del chiamante di usare un predicato in modo riflessivo e da' luogo alla trasformazione suddetta. La notazione lessicale e' analoga a quella delle formule atomiche, preceduta pero' dal carattere "~". Ad esempio, una chiamata del tipo

~ $p(arg_1 \ldots arg_n)$ verra' trasformata per effetto del reflecter diventando

$p('[arg_1 \ldots arg_n]$ "ambiente" "albero_di_ricerca")

e verra' quindi elaborata al livello del processore. E' opportuno osservare che, in relazione al passaggio di livello e quindi al cambiamento del "punto di vista" del programma, i parametri attuali vengono sostituiti da un riferimento alla lista degli argomenti stessi.

Un importante risultato riportato in [Ghi87a] e' la dimostrazione della possibilita' di definire il metaprocessore circolare del Logic con le caratteristiche di tail–ricorsivita' necessarie per l' implementazione finita della torre riflessiva. Vale la pena di notare che rispetto al 3–LISP le difficolta' sono maggiori perche', a causa del non determinismo, la proprieta' di tail–ricorsivita' per un linguaggio logico e' piu' complessa da ottenersi che per un linguaggio funzionale.

3. Oltre il Logic

Il motivo principale di interesse per l' architettura logico–riflessiva e' che essa unisce la chiarezza concettuale ad una notevole flessibilita'. Queste caratteristiche la candidano a costituire una base per la realizzazione di sistemi complessi per la rappresentazione ed uso di metaconoscenza. Per suffragare queste affermazioni e'

utile innanzitutto confrontare le caratteristiche dell' architettura con i requisiti enunciati in [Aie86].

1. Se la torre riflessiva e' relativa ad un linguaggio L, il linguaggio di rappresentazione della conoscenza e' lo stesso a tutti i livelli, ed e' appunto L.

2. La torre riflessiva si basa sul metaprocessore circolare del linguaggio stesso. Ad ogni livello e' percio' disponibile appieno l' apparato deduttivo del formalismo.

3. E' possibile definire un meccanismo di "naming", come dimostra l' esempio del Logic. Appare inoltre realistico definire meccanismi piu' evoluti sulla base della semplice referenziazione.

4. Il metaprocessore circolare di un linguaggio logico non e' altro che un' assiomatizzazione (nel linguaggio) della relazione di dimostrabilita' relativa al linguaggio stesso. Il metaprocessore puo' essere invocato a qualunque livello e in qualunque punto del programma.

   E' opportuno al riguardo mettere in evidenza la differenza con altri approcci. La relazione di dimostrabilita' puo' essere rappresentata nel linguaggio mediante un metainterprete [Bow82]: non esiste pero' garanzia di coerenza con l' interprete reale (che implementa effettivamente la dimostrabilita').

   Oppure l' implementazione della dimostrabilita' puo' essere resa utilizzabile al programma mediante un particolare predicato, che attiva una istanza dell' interprete ([Aie86], [Bow85]); in questo caso, il vincolo della coerenza e' rispettato ma la rappresentazione della dimostrabilita' e' al di fuori del linguaggio.

   L' architettura basata sul metaprocessore soddisfa entrambi i requisiti: rappresentazione della dimostrabilita' nel linguaggio, e insieme connessione causale con l' interprete reale.

5. I principi di riflessione sono realizzati mediante la riflessione procedurale. Vale la pena di ricordare che essa garantisce la connessione causale fra livelli, nel senso di preservare la coerenza fra lo stato di ciascun livello e la sua rappresentazione.

Per quanto riguarda la flessibilita' dell' architettura, e' facile convincersi del fatto che la riflessione procedurale consente di realizzare facilmente tutti i predicati extra–logici e meta–logici presenti nelle attuali implementazioni del Prolog, incluse la "negation as failure" e la generazione di lemmi (esempi in proposito sono riportati nei lavori citati a proposito del Logic). Al di la' delle estensioni

linguistiche, l' architettura puo' esplicare appieno le sue potenzialita' definendo opportunamente le strutture dati da rendere esplicite all' atto della riflessione.

La rappresentazione dell' ambiente puo' essere definita in modo piu' articolato rispetto alla semplice lista di associazioni del Logic. Le esigenze di cui tenere conto possono essere di diverso tipo, quali ad esempio: metodi di indicizzazione di clausole (anche allo scopo di implementare in modo efficiente predicati per la definizione di teorie, di cui si discutera' piu' avanti) oppure generazione e memorizzazione di lemmi positivi e negativi.

E' possibile esplicitare all' atto della riflessione (oltre ad ambiente e albero di dimostrazione) le strategie per il controllo dell' inferenza, ossia la strategia di scelta della clausola da selezionare fra le possibili candidate all' unificazione (nondeterminismo di tipo "OR") e della sottometa da considerare in una congiunzione (nondeterminismo di tipo "AND"). Lo scopo e' permettere al programma di ridefinire tali strategie, e di utilizzare strategie diverse in contesti diversi.

Sulla base della rappresentazione dell' albero di dimostrazione, il controllo dell' inferenza puo' andare oltre, in quanto si possono definire strategie diverse da quella "depth–first", o metodi per il "backtraking intelligente". Naturalmente e' difficile pensare di demandare all' utente medio la programmazione di strategie complesse. Piuttosto, un sistema logico–riflessivo disporra' di una biblioteca di strategie diverse, e di un insieme di predicati di sistema per modificare la strategia corrente. In tal modo l' utente potra' avvalersi di un insieme di costrutti di "alto livello" basati sulla riflessione, ma il cui uso non richiede la conoscenza del modello e delle strutture dati del processore.

Piu' oltre, e' in corso di indagine la possibilita' di definire (e realizzare mediante riflessione procedurale) estensioni al linguaggio delle clausole di Horn, volte ad introdurre esplicitamente aspetti metalinguistici. Lo scopo e' di fare in modo che una teoria possa includere indifferentemente regole e metaregole, queste ultime concernenti la definizione di proprieta' o la formulazione di asserzioni sulla conoscenza di livello oggetto, e sulle modalita' (di aspetti) della risoluzione dei problemi. In questo senso le asserzioni metateoriche influenzano il comportamento deduttivo della teoria: un simile punto di vista a questo proposito viene presentato in [Nar86], proponendo di definire al metalivello strategie di controllo della dimostrazione. L' argomento delle estensioni linguistiche non viene approfondito in questo lavoro, dove si intende principalmente evidenziare la versatilita' della riflessione procedurale come strumento di base. A questo scopo al paragrafo seguente si discuitera' la possibile realizzazione di diverse versioni del predicato *demo*, esemplificandone poi le potenzialita' di utilizzo.

4. Definizione di teorie, e predicato demo

L' obbiettivo di questo paragrafo e' mostare come un' architettura logico–riflessiva – imperniata sul solo meccanismo della riflessione procedurale – possa costituire la base per la realizzazione di un sistema logico dotato di strumenti evoluti di ausilio alla programmazione, tutti realizzati nell' ambito del modello mediante predicati riflessivi.

Si e' in precedenza osservato che nell' architettura esiste gia' una versione operazionale del predicato *demo*. Le limitazioni rispetto alla proposta originale di Bowen e Kowalski [Bow82] sono di due tipi.

- Il predicato *demo* realizzato mediante il metaprocessore del linguaggio esclude talune prestazioni ipotizzate in [Bow82], e cioe' determinare la teoria dalla quale e' derivabile una certa proposizione, o viceversa l' insieme delle proposizioni derivabili da una formula. In sostanza, la formula da dimostrare e l' ambiente di dimostrazione devono essere definiti in ingresso.
- L' ambiente nel quale la proposizione viene dimostrata e' quello dell' interprete del livello corrente. L' intento e' invece, come e' largamente condiviso nelle principali proposte apparse in letteratura ([Bow82], [Aie86], [Bow85]), quello di disporre di un predicato che sia almeno della forma

  demo(Teoria, Proposizione, Risposta)

  dove il primo parametro rappresenta la teoria logica rispetto alla quale si vuole dimostrare la proposizione data come secondo parametro. La risposta puo' essere il valore booleano *true* o *false* (a seconda che la dimostrazione riesca o meno) o, in caso di successo, una lista contenente oltre all' atomo *true* i valori di unificazione delle variabili che compaiono nella meta.

Il primo inconveniente non e' eliminabile in quanto e' legato al modello sul quale il Logic e' basato, ossia l' architettura riflessiva composta sulla definizione operazionale del linguaggio. Rispetto al secondo punto, si esemplifichera' di seguito il modo di realizzare il predicato *demo* generalizzato senza modificare l' interprete, mediante riflessione procedurale.

Il principale meccanismo da implementare nell' ottica della realizzazione di *demo* e' relativo alla definizione delle teorie. In particolare e' necessario un predicato del tipo

definisci_teoria(Contenuto, Nome).

L' implementazione richiede un predicato riflessivo il quale, tramite uno spostamento di livello verso l' alto, acceda all' ambiente corrente per creare il legame Contenuto–Nome. Il contenuto puo' essere specificato in ingresso come elenco di

clausole, o come risultato della manipolazione di teorie esistenti (unione di teorie, aggiunta o sottrazione ad una teoria di clausole specificate). In ogni caso, e' possibile evitare ridondanze collegando il nome al contenuto mediante meccanismi di riferimento, come proposto in [Bow85] e in [Bac87], dove viene svolto uno studio approfondito del problema.

La manipolazione di teorie puo' comportare, oltre che aggiunta o rimozione di clausole, anche l' adattamento delle teorie stesse al particolare uso, tipicamente modificando il nome dei predicati e/o cambiandone le modalita' di utilizzo. Un meccanismo del genere risulterebbe particolarmente orientato a problemi di riusabilita' del software, consentendo di definire operazioni di trasformazione su moduli preesistenti. Si pensi tipicamente all' esigenza di adattare la definizione di strutture dati di natura generale (matrici, code, alberi, ecc.) alla soluzione di problemi particolari. Un operatore di trasformazione puo' essere implementato tramite un predicato riflessivo che crei le clausole della nuova teoria copiando e modificando secondo le specifiche ricevute in ingersso quelle della teoria (o delle teorie) di partenza.

Una funzionalita' che puo' essere utile integrare in un sistema logico e' la valutazione parziale [Tak86]. Disponendo di un algoritmo di valutazione parziale, e' possibile rendere accessibile al programma un predicato del tipo

componi(Meta_interprete, Teoria_oggetto, Teoria_risultante).

mediante un predicato riflessivo che attivi una istanza dell' interprete per eseguire la valutazione, per poi creare la nuova teoria corrispondente al risultato.

A sua volta, l' implementazione di *demo* comporta semplicemente uno shift verso l' alto per estrarre dall' ambiente la teoria specificata al primo argomento, creando un nuovo ambiente su cui attivare un' istanza dell' interprete per dimostrare la proposizione specificata. Si puo' inoltre estendere la definizione di *demo* con prestazioni addizionali. In questa sede se ne propongono due in particolare, le cui possibilita' di utilizzo verranno discusse al paragrafo successivo in relazione agli esempi.

Una prima caratteristica di cui si intende proporre l' integrazione in un sistema orientato all' utilizzo di metaconoscnza e' la possibilita' di chiamate del tipo:

$$demo(T + (Hp_1 \vee \ldots \vee Hp_n),\ Prop,\ Risp).$$

Il significato e' equivalente alla disgiunzione delle n chiamate di *demo* su ciascuna delle teorie ottenute dall' unione di T con $Hp_1 \ldots Hp_n$ rispettivamente, ossia

$$demo(T + Hp_1,\ Prop,\ Risp) \vee \ldots \vee demo(T + Hp_n,\ Prop,\ Risp).$$

La risposta e' affermativa se almeno una delle chiamate ha successo. In tal caso il parametro di uscita conterra', oltre all' atomo *true*, la soluzione relativa a ciascuna delle ipotesi che ha avuto successo. Se tutte le ipotesi hanno avuto successo, si puo' restituire il solo atomo *true*, in quanto le diverse ipotesi risultano ininfluenti per l' esistenza di una soluzione.

La seconda caratteristica proposta e' una variazione rispetto al caso precedente, per considerare i casi in cui le ipotesi alternative siano rappresentate dalle diverse clausole di una stessa teoria. Tipicamente, come verra' illustrato negli esempi, la teoria in questione conterra' il completamento di una relazione definita precedentemente in modo parziale. La forma della chiamata, il cui significato e' illustrato di seguito, puo' essere ad esempio:

$$demo(T + \vee Tc[\delta],\ Prop,\ Risp).$$

L' espressione sintattica $\vee Tc[\delta]$ indica il particolare uso che si vuole fare della teoria Tc. Il simbolo δ indica una lista di associazioni variabile–valore, specificata per specializzare la teoria Tc rispetto alle variabili in essa contenute. In particolare, se $C_1 \ldots C_n$ sono le clausole componenti la teoria Tc, e se $C_i[\delta]$ e' il risultato dell' applicazione della sostituzione δ alla clausola C_i, la chiamata sopra si puo' ricondurre al caso precedente, essendo equivalente a

$$demo(T + (C_1[\delta] \vee \ldots \vee C_n[\delta]),\ Prop,\ Risp).$$

Per finire e' opportuna qualche considerazione in merito a questioni di efficienza. Il sovraccarico introdotto manipolando teorie e invocando *demo* contiene in primo luogo un fattore legato al numero di passaggi di livello che si determinano nei predicati riflessivi che implementano tali meccanismi. Il costo di uno shift up o down dovrebbe prevedibilmente essere una quantita' costante, valutabile in relazione alla particolare implementazione del level–shifting processor. C' e' poi il costo relativo alla creazione, modifica e reperimento di teorie. Questo puo' essere minimizzato come accennato sopra tramite opportune strategie di implementazione, in modo da risultare accettabile in relazione ai vantaggi conseguiti in termini di capacita' espressive.

5. Esempi e considerazioni sull' uso di metaconoscenza

Allo scopo di chiarire l' utilita' dei predicati definiti al paragrafo precedente, ed in particolare di quelli meno usuali, si prendera' in considerazione un problema campione in relazione al quale esaminare possibili metodi di soluzione. Il problema e' definito in modo da presentare aspetti di conoscenza incompleta, per dare l' occasione di mostrare i vantaggi derivanti dalla definizione e uso di metaconoscenza.

La formulazione del problema e' la seguente. Si considerino tre blocchi adiacenti, a, b e c, il primo di colore rosso, il secondo di colore non noto ed il terzo di colore verde. Si descriva questa situazione mediante una teoria logica specificata in clausole di Horn. Si definisca quindi un metodo per determinare se esista un blocco verde immediatamente a destra di uno non verde. Come si vede il problema e' banale, non altrettanto banale e' invece specificarne la soluzione in clausole di Horn.

Le clausole riportate sotto formalizzano lo scenario ipotizzato nella formulazione del problema.

```
blocco(a).
blocco(b).
blocco(c).

adiacente(a, b).
adiacente(b, c).

colore(a, rosso).
colore(c, verde).
```

Una possibile definizione della soluzione e' la seguente:

```
blocco(X), blocco(Y), adiacente(X, Y), non_verde(X), verde(Y).
```

dove si sia definito

```
non_verde(X) :-   colore(X, C),
                  C \== verde.

verde(X) :-       colore(X, verde).
```

E' evidente che, rispetto alla strategia di risoluzione standard, questa formulazione del problema non e' adeguata. Infatti entrambe le possibili soluzioni, ossia le coppie [X/a, Y/b] e [X/b, Y/c] contengono il blocco b, il cui colore non e' noto, cosa che determina il fallimento della dimostrazione; in particolare, nel primo caso il fallimento e' in corrispondenza della sottometa colore(b, verde); nel secondo, della sottometa colore(b, C).

Si supponga ora di definire una teoria, scenario, che contenga le definizioni di base, ed una teoria, classi, che contenga le definizioni di verde(_) e non_verde(_). La soluzione del problema puo' essere determinata tramite la

seguente chiamata di *demo* (se $\mathfrak{g}$ e' un nome per la congiunzione di mete riportata sopra, che definisce i requisiti della soluzione):

```
unione(scenario, classi, Teoria_base),
demo(Teoria_base + ("verde(b)" v "non_verde(b)"), g, Risposta).
```

In questo caso la risposta e' semplicemente *true*, perche' i requisiti sono soddisfatti sotto tutte le condizioni. Emerge quindi come nello scenario specificato la conoscenza del colore del blocco b sia irrilevante per la soluzione del problema. Se invece, ad esempio, nello scenario iniziale anziche' "colore(c, verde)." vi fosse stato "colore(c, blu).", si sarebbe ottenuta una risposta condizionata all' ipotesi valida, ossia del tipo "Supponendo verde(b), [X/a, Y/b]". In sostanza, i vantaggi ottenuti consistono nella possibilita' di esprimere la disgiunzione (nella forma "vale questo o quello"), di ottenere risposte condizionate, e di appurare se determinate conoscenze siano o meno essenziali in un certo contesto.

Considerando il problema da un altro punto di vista, si supponga di definire una teoria, colori, che specifichi le possibilita' ammesse in merito al colore degli oggetti. Nel caso presente le possibilita' sono espresse dalle asserzioni seguenti.

```
colore(X, blu)
colore(X, giallo).
colore(X, rosso).
colore(X, verde).
```

Un altro modo possibile di ottenere la soluzione e':

```
unione(scenario, classi, Teoria_base),
demo(Teoria_base + vcolori[X/b], g, Risposta).
```

La differenza con l' approccio seguito in precedenza e' che in questo caso, piuttosto che formulare delle ipotesi precise, si preferisce limitarsi ad elencare le possibilita' completando la definizione della relazione colore(_, _), e demandando al sistema il compito di verificare tutte le possibilita'. Il risultato ottenuto e' in questo particolare caso lo stesso.

In sostanza le due soluzioni esprimono due modi diversi di trattare conoscenza incompleta. Uno si basa sulla formulazione di ipotesi (eventualmente arbitrarie) in merito alle situazioni che possono presentarsi. L' altro consiste nel precisare i casi possibili; rispetto al precedente e' in un certo senso piu' rigoroso, ma e' al tempo stesso piu' dispendioso, perche' richiede un esame di tutte le possibilita'. Entrambi presuppongono l' intervento del programmatore per definire le ipotesi o il completamento. Rimane fuori la possibilita' che il sistema tenti di inferire dalla conoscenza in suo possesso le ipotesi possibili.

I meccanismi presentati non trovano applicazione soltanto nel trattamento di conoscenza incompleta, ma in generale in tutte le forme di ragionamento ipotetico, in cui si voglia verificare che cosa accada in circostanze diverse, o quali siano le conseguenze di determinati presupposti. Vale la pena di osservare che la verifica di ipotesi nel contesto della chiamata di *demo* garantisce al programma il requisito di base del ragionamento ipotetico (e quindi della "riflessione" nel senso psicologico del termine), ossia la capacita' di osservare i propri processi di ragionamento da un punto di vista distaccato. E' infatti possibile formulare diverse ipotesi di comportamento e/o di interpretazione di cio' che e' noto, esaminarne le implicazioni, metterle a confronto, e quindi trarre le conclusioni e pianificare di conseguenza il corso successivo dell' elaborazione.

6. Sviluppi futuri

E' stata avviata l' attivita' di implementazione di un prototipo di architettura logico–riflessiva con le caratteristiche precedentemente discusse. I mezzi impiegati sono la stazione di lavoro IBM 6150 con sistema operativo AIX, e il linguaggio Prolog. Contemporaneamente, si intende continuare l' approfondimento sul tema dell' utilizzo della metaconoscenza e della riflessione per migliorare le capacita' espressive e deduttive di un sistema logico. L' utilizzo del Prolog nello sviluppo del prototipo ha due diverse finalita': la prima e' legata all' esperienza di utilizzo di un linguaggio logico nello sviluppo di un sistema complesso; la seconda e' legata alla possibilita' di ottenere in tempi brevi versioni incomplete ma funzionanti del prototipo, in modo da integrare utilmente le due linee di lavoro.

Ringraziamenti

Gli autori desiderano ringraziare il Prof. Giovanni Degli Antoni per aver evidenziato le implicazioni dei lavori di Smith e per il costante supporto all' attivita' di ricerca, e il Dott. Luca Spampinato per averne effettuato le prime estensioni e realizzazioni e per le utili discussioni sugli argomenti trattati in questo lavoro.

Riferimenti

[Aie86] L. Aiello, C. Cecchi, D. Sartini, *Representation and Use of Meta–knowlwdge*, Proceedings of the *IEEE*, vol. 74, n. 10, October 1986, pp. 1304 – 1321.

[All78] J. Allen, *Anatomy of LISP*, McGraw–Hill, 1978.

[Bac87] H. Bacha, *Meta–level Programming: a Compiled Approach*, Proceedings of the 4th International Conference on Logic Programming, Melbourne, Australia, 1987, pp. 394–410.

[Bow82] K. A. Bowen, R.A. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming*, in K. L. Clark and S–A. Tarnlund (eds.), *Logic Programming*, Academic Press, London, 1982.

[Bow85] K. A. Bowen, *A Meta–level Extention of Prolog*, Proceedings of 1985 IEEE Symposium on Logic Programming, Boston, USA, 1985, pp. 669–675.

[DeR84] J. de Rivieres, B. C. Smith, *The Implementation of Procedurally Reflective Languages*, Xerox PARC ISL–4, Palo Alto, 1984.

[Fur88] F. Furlan, G. A. Lanzarone, *Prolog: linguaggio e metodologia di programmazione logica*, Franco Angeli ed., Milano, 1988.

[Gen87] M. R. Genesereth, N. J. Nillson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers inc., Los Altos, California, 1987.

[Ghi87] R. Ghislanzoni, L. Spampinato, G. Tornielli, *The Reflective Alice System*, Universita' degli Studi di Milano, *Rapporto Tecnico*, 1987.

[Ghi87a] R. Ghislanzoni, L. Spampinato, G. Tornielli, *Reflection as a Tool for Integration: an Exercise in Procedural Introspection*, Proc. of 10th International Joint Conference on Artificial Intelligence, Milano, 1987.

[Llo84] J.W. Lloyd, *Foundation of Logic Programming*, Springer–Verlag, Berlin, 1984.

[Mae87] P. Maes, *Computational Reflection*, Artificial Intelligence Laboratory Technical Report 87–2, University of Bruxelles, 1987.

[Nar86] D. Nardi (1986), *Evaluation and Reflection in FOL*, In: *Meta–level Architectures and Reflection*, P. Maes and D. Nardi Eds., North–Holland, Amsterdam, 1987.

[Smi84] B. C. Smith, *Reflection and Semantics in LISP*, Xerox PARC ISL–5, Palo Alto, 1984.

[Smi86] B.C. Smith, *Varieties of Self–Reference*, in Theoretical Aspects of Reasoning about Knowledge, *Proc. of the 1986 Conference*, Ed. Halpern J. Morgan Kauffmann.

[Tak86] A. Takeuchi, K. Furukawa, *Partial Evaluation of Prolog Programs and its Application to Meta Programming*, Elsevier Science Publishers B. V. (North Holland), *IFIP*, 1986.

[Wie86] B. J. Wielinga, J. A. Breuker, *Models of Expertise*, Proc. of 7th European Conference on Artificial Intelligence, Brighton, UK, 1986.