Meta-Programmazione Logica in 'Log

I. Cervesato - G.F. Rossi

Dipartimento di Matematica e Informatica,
Universita' degli Studi di Udine,
Via Zanon, 6 - 33100, Udine.

Sommario

In questo lavoro viene mostrato come dotare un linguaggio per la programmazione logica di capacita' di meta-programmazione in modo da conservare la semantica logica del linguaggio pur senza pregiudicare l'efficienza di una sua implementazione. L'estensione consiste essenzialmente nella definizione di un opportuno **schema di naming** in grado di associare **due** diverse meta-rappresentazioni a **tutte** le entita' sintattiche del linguaggio tramite cui poterle manipolare a diversi livelli di dettaglio. Il linguaggio risultante, chiamato **'Log**, e' in grado di rimpiazzare tutti i meta-predicati built-in di Prolog con una loro definizione logica data nel linguaggio stesso, offrendo in piu' nuove possibilita' di meta-programmazione derivanti dalla capacita' di trattare interi programmi al meta-livello. Il linguaggio risulta inoltre implementabile in modo efficiente ed e' dotato di una semantica logica precisa, definita come semplice estensione di quella standard.

1. Introduzione

E' noto che le implementazioni di Prolog forniscono solitamente alcune capacita' di meta-programmazione sotto forma di predicati built-in (come *var/1*, *clause/2*, *call/1*, ecc.) e che queste risultano molto utili in svariate applicazioni (ad es. nella realizzazione di meta-interpreti [StS86] e [StL88] o nella costruzione di sistemi IA [ACS86]). E' tuttavia altrettanto noto quanto poco soddisfacenti siano tali "capacita' meta" dal punto di vista della loro semantica logica.

Un primo tentativo di affrontare il problema della meta-programmazione in un linguaggio di programmazione logica in modo sistematico e' quello fatto da Bowen e Kowalski in [BoK82]. Numerosi sono stati successivamente i lavori che hanno preso spunto da questa proposta. Relativamente pochi sono invece gli studi che, seguendo l'approccio delineato da questi autori, si sono posti come obiettivo la realizzazione di un linguaggio logico concreto con capacita' di meta-programmazione simili a quelle gia' disponibili in Prolog, ma in grado di soddisfare requisiti quali:

- poter trattare al meta-livello **qualsiasi** entita' sintattica del linguaggio, dai programmi ai simboli;
- permettere un'**implementazione efficiente**, sia in termini di tempo di esecuzione che di occupazione di memoria;
- possedere una **semantica logica** per tutto il linguaggio (compresi i meta-predicati);

Lavoro parzialmente finanziato da C.N.R. - Progetto Finalizzato "Sistemi Informatici e Calcolo parallelo" e da M.U.R.S.T. 40%.

- offrire strumenti di meta-programmazione che siano sufficientemente naturali e **semplici da usare**.

Tra le proposte che si sono mosse in questa direzione, vanno senz'altro ricordati MetaProlog [Bow85], [BoW85] e, molto piu' recentemente, il linguaggio Gödel [BHL90]. Al momento attuale, non sembra che nessuna di queste proposte soddisfi appieno i requisiti sopra esposti.

Obiettivo di questo lavoro e' di definire un'estensione ad un linguaggio di programmazione logica che, muovendosi nella direzione delle proposte sopra citate, sia in grado di soddisfare tutti i requisiti appena elencati. L'estensione consiste essenzialmente nella definizione di un opportuno **schema di naming** in grado di associare alle diverse entita' sintattiche del linguaggio un nome tramite cui poterle manipolare al meta-livello. L'idea fondamentale della nostra proposta e' di avere **due** diverse meta-rappresentazioni per ogni entita' sintattica del linguaggio, nonche' un operatore con cui passare da una meta-rappresentazione all'altra. Il linguaggio risultante, denominato '**Log** (e pronunciato "quote log") presenta capacita' di meta-programmazione paragonabili a quelle di Prolog (e per certi aspetti superiori), senza per cio' dovere rinunciare ad una semantica rigorosamente logica e ad una possibile implementazione efficiente.

L'idea della doppia meta-rappresentazione per una stessa entita' sintattica era gia' stata applicata, da un punto di vista piu' pragmatico e limitatamente ai programmi quali uniche entita' sintattiche considerate, alla definizione delle capacita' meta del linguaggio **EnvProlog**, un'estensione di Prolog orientata alla costruzione di ambienti di programmazione Prolog [MaR88] [Ros89] Ros90]. Nel lavoro attuale, invece, il punto di partenza e' un linguaggio di programmazione logica "puro" e lo schema di naming viene esteso a tutte le entita' sintattiche del linguaggio, precisando sintassi e semantica (dichiarativa ed operazionale) del linguaggio risultante.

Nel capitolo 2, verranno brevemente presentate le principali caratteristiche di 'Log. I capitoli 3 e 4 descrivono la sintassi del linguaggio, in particolare quella relativa alle due meta-rappresentazioni. La funzione e l'utilita' di questa doppia meta-rappresentazione sono illustrate nei capitoli 5 e 6. Il successivo capitolo 7 descrive invece le primitive disponibili in 'Log per la manipolazione dei programmi. I principali risultati teorici relativi alla semantica di 'Log sono esposti nel capitolo 8. Infine, il capitolo 9 confronta 'Log con altre analoghe proposte.

2. Caratteristiche generali di 'Log

Ad ogni entita' sintattica di 'Log sono associate due **meta-rappresentazioni**, ossia due termini ground che la descrivono univocamente e con cui e' possibile manipolarla al meta-livello.

Ogni oggetto sintattico e' innanzitutto descritto da una costante, ad esso sintatticamente simile, detta il suo **nome**. Gli oggetti dotati di nome in 'Log sono i programmi, le clausole, i termini, i simboli ed i caratteri. Il fatto di dare un nome anche ai caratteri puo' inizialmente sorprendere in quanto essi non fanno propriamente parte della classe dei costrutti sintattici di un programma logico. L'accesso ai caratteri, sia pure mediante il loro nome, risulta pero' estremamente comodo; e' ad esempio grazie a questa possibilita' che riusciremo a scrivere nel capitolo 8 una definizione semanticamente accettabile del predicato *var/1*.

Ogni entita' sintattica propria - non i caratteri pertanto - ha, oltre ad un nome, anche una **rappresentazione strutturale**, ossia un termine ground che ne descrive la struttura in termini dei

nomi delle entita' sintattiche componenti. Nome e rappresentazione strutturale sono due descrizioni di un oggetto sintattico a due diversi livelli di dettaglio.

Anche i nomi sono delle entita' sintattiche in 'Log, ed hanno pertanto anch'essi un nome ed una rappresentazione strutturale. 'Log offre quindi la possibilita' di definire una successione di **meta-livelli** di profondita' arbitraria. Nella maggior parte delle applicazioni pero', si fara' riferimento solamente ad un livello oggetto e ad un meta-livello.

Per ogni oggetto sintattico, e' possibile passare dal suo nome alla sua rappresentazione strutturale e viceversa mediante l'operatore <=>*/2*, scritto infisso. L'esatto comportamento di <=> verra' descritto nei capitoli 5 e 7.

In 'Log, non esiste, ne' e' possibile definire, nessun operatore di riflessione che permetta di passare da un oggetto sintattico ad una sua meta-rappresentazione, o viceversa. In questo, 'Log si differenzia da altre proposte quali Reflective Prolog [CoL89] o R-Prolog* [Sug90], in cui viceversa si assume che un tale operatore sia disponibile (anche se non visibile all'utente). In questo senso, i livelli di meta-programmazione sono separati in 'Log. Ad ogni livello, le entita' sintattiche dei livelli sottostanti sono visibili solamente attraversi i loro nomi. Le variabili non fanno eccezione a questa regola.

Nomi, rappresentazioni strutturali e <=> sono sufficienti alla definizione di ogni meta-predicato di Prolog. E' cosi' possibile recuperare questi predicati in 'Log senza pregiudicare l'interpretazione dichiarativa di un programma. Un'implementazione concreta di 'Log dovrebbe comunque fornire anche una realizzazione di piu' basso livello di molti di questi meta-predicati per ragioni di praticita' ed efficienza.

Un altro aspetto peculiare di 'Log, ortogonale questo alle sue caratteristiche di linguaggio di meta-programmazione, e' di fondarsi su una logica del prim'ordine da cui e' stata rimossa la separazione tra i simboli predicativi ed i simboli funzionali. Le formule atomiche che costituiscono un programma 'Log sono semplicemente termini, eventualmente variabili.

3. Nomi

Un programa 'Log e' sintatticamente simile ad un programma Prolog. Per ragioni di brevita', ci limiteremo a descrivere soltanto quegli aspetti che risultano significativamente differenti dal caso standard.

Diversamente dalla usuale definizione sintattica di programma logico, un programma 'Log non e' costruito a partire da insiemi di simboli funzionali, predicativi e di variabili. Ha invece come base un **alfabeto**, che indicheremo con Σ, costituito da **caratteri**. Σ e' essenzialmente utilizzato nella costruzione dei simboli. E' cosi' possibile scomporre un simbolo nei caratteri che lo costituiscono, trattandoli come entita' sintattiche del linguaggio. Σ e' partizionato in un alfabeto maiuscolo, uno minuscolo ed uno riservato, nell'intento di seguire, ove possibile, le convenzioni sintattiche del Prolog standard.

'Log distingue tre tipi di **simboli**: oltre agli usuali simboli funzionali e alle variabili, vi e' una terza categoria sintattica di base, i nomi. La definizione di nome richiede che vengano introdotte prima le nozioni di programma, clausola e termine.

I **termini** vengono definiti nel solito modo ricorsivo a partire dai simboli (fra cui, ricordiamo, vi sono i nomi). In particolare, un nome e' un termine. La definizione di **clausola** (di Horn) si distingue da quanto descritto comunemente per il fatto che le formule atomiche di un programma 'Log sono generici termini. E' infatti gia' stato detto che 'Log non distingue tra simboli predicativi e simboli di funzione. Come al solito, il suffisso :- sara' opzionale per un fatto, tranne che nelle meta-rappresentazioni di un programma. Infine, un **programma** 'Log e' una sequenza di clausole separate da punti. Nel contesto di 'Log, il concetto di sequenza e' piu' appropriato del solito concetto di insieme.

A questo punto, e' possibile dare la seguente definizione di **nome**:

- se P e' un programma, allora {P} e' un nome detto il **program name** di P;
- se C e' una clausola, allora 'C' e' un nome detto il **clause name** di C;
- se t e' un termine, allora 't' e' un nome detto il **term name** di t;
- se s e' un simbolo, allora `s` e' un nome detto il **symbol name** di s;
- infine, se c e' un carattere, allora %c e' un nome detto il **character name** di c.

Ad ogni simbolo e' associata, come in Prolog, un'arita', nulla nel caso delle variabili e dei nomi.

Si noti che una stessa espressione sintattica puo' avere piu' nomi in dipendenza del contesto in cui e' utilizzata. Consideriamo ad esempio la stringa composta dal solo carattere *c*. Secondo la sintassi di 'Log, *c* e' sia un carattere che un simbolo che un termine. Pertanto, il character name *%c*, il symbol name *`c`* e il term name *'c'* sono tutte meta-rappresentazioni di questa medesima stringa.

Osserviamo inoltre che viene utilizzato lo stesso carattere (') per formare i nomi dei termini e delle clausole. E' tuttavia possibile disambiguare i term name dai clause name per la presenza della sottostringa :- nei secondi in una posizione inappropriata per i primi. Pertanto, `'append(X,[],X):-'` e' chiaramente un clause name.

A titolo esemplificativo, consideriamo la solita procedura *append/3*, che concatena due liste. Il suo program name e':

```
{append([],X,X):-.
 append([A|X],Y,[A|Z]):-append(X,Y,Z).}
```

ove la notazione per le liste si suppone definita nel solito modo.

I nomi sono anch'essi dei simboli e quindi hanno a loro volta un nome. Pertanto il nome del character name *%c*, del symbol name *`pippo`* e del term name *'Alfa'* sono rispettivamente *`%c`*, *``pippo``* e *`'Alfa'`*. E' altrettanto lecito utilizzare il nome di un clause name o di un program name. 'Log non pone limiti al livello di naming; e' pertanto permesso parlare del nome del nome del nome del nome del simbolo pippo, che si scrive *````pippo````*. Si noti che, essendo i nomi dei simboli, e' evidente che i due term name *'p(X)'* e *'p(Y)'* non sono unificabili, mentre e' possibile unificare la variabile *X* con *'f(X)'*, dando origine alla sostituzione *{X/'f(X)'}*, senza nessun veto da parte dell'occur check.

4. Rappresentazioni strutturali

Come gia' accennato nel capitolo 2, 'Log associa ad ogni forma sintattica composita (e cioe' simboli, termini, clausole e programmi) una seconda meta-rappresentazione - detta

rappresentazione strutturale - costituita da un termine ground che esprime la struttura dell'oggetto denotato in funzione dei **nomi** delle entita' sintattiche che lo compongono.

La rappresentazione strutturale ha normalmente la forma di una lista di nomi. Fanno eccezione, le clausole, per le quali si utilizza il simbolo riservato *clause/2*, ed i termini, la cui definizione ricorsiva richiede liste annidate di symbol name. Oltre alla notazione con liste, o **esplicita**, 'Log mette a disposizione una notazione sintattica alternativa, detta **sintetica**, piu' facile da usare per l'utente.

Definiamo ora in modo preciso la rappresentazione strutturale delle diverse entita' sintattiche di 'Log. Viene prima data la rappresentazione sintetica e poi quella esplicita.

- Se $P = C_1.C_2. \ldots C_n.$ e' un programma, allora
  $\{\{C_1.C_2. \ldots C_n.\}\} = ['C_1','C_2',\ldots,'C_n']$
  e' il **program structure** di P. Il program structure del programma vuoto e' {{}}.

- Se $C = A :- A_1,A_2,\ldots,A_n$ e' una regola o un fatto, allora
  $"A:- A_1,A_2,\ldots,A_n" = clause('A',['A_1','A_2',\ldots,'A_n'])$
  e' il **clause structure** di C.

- Se $t = f(t_1,t_2,\ldots,t_n)$ e' un termine con n>0, allora (ricorsivamente)
  $"f(t_1,t_2,\ldots,t_n)" = [`f`,"t_1","t_2",\ldots,"t_n"];$
  e' il **term structure** di t. Nel caso in cui t = s sia una costante, una variabile o un nome, il term structure di t e' dato da
  "t" = `s`.

- Se $s = c_1c_2\ldots c_n$ e' un simbolo, allora
  $^c_1c_2\ldots c_n^ = [\%c_1,\%c_2,\ldots,\%c_n]$
  e' il **symbol structure** di s.

Mentre i nomi sono simboli atomici, le rappresentazioni strutturali sono termini ground compositi. In questo contesto, termini aventi l'aspetto di rappresentazioni strutturali se non per la presenza di variabili (del meta-livello) al loro interno sono da intendersi come rappresentazioni strutturali **non completamente specificate**. Ad esempio,

```
[`f`,X]
```

non e' un term structure; se pero' la variabile X viene istanziata al symbol name *`a`*, o al term structure *"g(a,b)"*, si ottiene una rappresentazione strutturale compiuta: *"f(a)"*, oppure *"f(g(a,b))"* rispettivamente. Occorrenze di variabili del meta-livello in una rappresentazione strutturale incompleta verranno chiamate **meta-variabili**, in contrasto con le variabili oggetto, che sono congelate nei nomi che compaiono in essa.

E' molto semplice costruire rappresentazioni strutturali contenenti meta-variabili rifacendosi alla notazione esplicita. Si vorrebbe pero' potere fare la stessa cosa anche con la notazione sintetica. Non vi e' nessun problema a fare cio' in un program structure; ad esempio:

```
{{p(X).Z.q(X, Y):-p(X),p(Y).}}
```

si puo' interpretare univocamente come

```
['p(X)',Z,'q(X, Y):-p(X),p(Y)']
```

essendo evidente che Z e' una meta-variabile dal contesto in cui essa appare. Procedere in modo analogo nel caso di clause e term structure non e' invece altrettanto semplice.[1] La notazione sintetica di questi oggetti richiede pertanto che le variabili di livello meta e quelle di livello oggetto siano distinte in modo esplicito, facendo precedere le ultime dal carattere #.[2] Pertanto,

```
clause('append([A|X],Y,[A|Z])',[B])
     = "append([#A|#X],#Y,[#A|#Z]):-B"
[`f`,`X`] = "f(#X)"
[`f`,X]   = "f(X)"
[F,`X`]   = "F(#X)"
[F,X]     = "F(X)"
```

sono tutte meta-rappresentazioni strutturali corrette. In questo caso, ovviamente, "f(X)" e "f(Y)" unificano producendo la sostituzione {X/Y}. Si noti invece che il goal *X = "f(X)"* dovebbe portare ad un fallimento in virtu' dell'occur check.

La solita notazione Prolog per indicare il resto di una lista puo' essere usata anche per le rappresentazioni strutturali incomplete in forma esplicita. E' poi facile estendere questa notazione alle rappresentazioni strutturali in forma sintetica. Ad esempio, possiamo definire il seguente term structure incompleto

```
[`f`,`a`|R].
```

a cui corrisponde la notazione sintetica

```
"f(a,|R)"
```

(si noti che il nome di un simbolo funzionale non tiene conto della sua arita'). Una sintassi simile e' disponibile anche per le rappresentazioni strutturali sintetiche di programmi, clausole e simboli. Come altro esempio, possiamo mostrare la definizione di un predicato che effettua la concatenazione di due program structure (cfr. [Ros90]):

```
appendPS({{}},P,P).
appendPS({{C|P1}},P2,{{C|P3}}):-appendPS(P1,P2,P3).
```

Riassumendo, 'Log si avvale del seguente schema di naming:

		nomi	rappresentazione strutturale sintetica
programmi:	P	{P}	{{P}}
clausole:	H:-B	'H:-B'	"H:-B"
termini:	t	't'	"t"
simboli:	s	`s`	^s^
caratteri:	c	%c	

Si osservi che la notazione sintetica e' stata descritta in riferimento ad un unico livello di meta-programmazione. Essa si estende in modo naturale ad un numero qualsiasi di livelli: il numero di #

---

[1] Non si e' ritenuto opportuno ne' utile introdurre una notazione sintetica anche per i symbol name parzialmente specificati

[2] Questa notazione e' stata preferita alla scelta duale di premettere # alle meta-variabili solamente in quanto sembra essere piu' naturale per l'utente.

davanti ad una variabile esprime il numero di livelli in cui addentrarci prima di trovare la variabile vera e propria, e non il suo nome.

5. Corrispondenza tra le due meta-rappresentazioni

In 'Log e' disponibile l'operatore <=>/2, detto di **destrutturazione**, per passare dal nome di un entita' sintattica alla corrispondente rappresentazione strutturale e viceversa. Il goal

X <=> **Y**

ha successo nei seguenti casi:

- se **X** e' un nome ed e' possibile istanziare (le meta-variabili che compaiono in) **Y** in modo tale da ottenere la rappresentazione strutturale corrispondente ad **X**;
- se **X** e' una variabile e **Y** e' la rappresentazione strutturale ground di un qualche oggetto sintattico.

Se sia **X** che **Y** contengono meta-variabili, la risoluzione di questo goal e' rimandata fino a quando una delle precedenti condizioni non venga raggiunta o nessun altro goal della computazione in corso possa essere selezionato. In quest'ultimo caso, questo termine viene restituito come parte della risposta calcolata.

Esempi di esecuzione di goal contenenti l'operatore <=> sono:

```
?- 'f(g(a),b,C)' <=> "f(A,b,#C)".          (1)
     A --> "g(a)".

?- N <=> "f(g(a),b,c)".                    (2)
     N --> 'f(g(a),b,c)'.

?- N <=> "f(A,b,#C)", A = "g(a)".          (3)
     A --> "g(a)",
     N --> "f(g(a),b,#C)".

?- N <=> "f(A,B,#C)", A = "g(a)".          (4)
     A --> "g(a)",
     N <=> "f(g(a),B,#C)".
```

Ritardare la risoluzione dei goal contenenti l'operatore <=> permette di non pregiudicare la lettura dichiarativa di un programma 'Log. L'ordine dei letterali in una clausola o in un goal (come ad es. (3)) e' irrilevante; la commutativita' di "," visto come operatore logico di congiunzione, e' pertanto conservata. Inoltre, anche se non e' possibile risolvere completamente la chiamata di <=> (perche' alcune variabili che compaiono in esso non sono state istanziate adeguatamente) la computazione non fallisce. Al contrario, <=> e' ritornato come parte della risposta calcolata nell'intento di esprimere un **vincolo** sui valori che possono assumere le variabili non ancora istanziate che compaiono in esso. Per esempio, in (4) esistono certamente istanze delle (meta-)variabili B ed N valide, ma non tutte lo sono. Una definizione piu' precisa dell'operatore <=> e della sua semantica verra' data nel capitolo 8.

6. Perche' due meta-rappresentazioni?

Vi sono diverse ragioni che giustificano la scelta fatta in 'Log di una doppia meta-rappresentazione per ciascuna entita' sintattica del linguaggio.

Una prima motivazione deriva da considerazioni sull'**uso** del linguaggio ed e' intesa ad ottenere una maggior **sinteticita' di notazione**, con la possibilita' di procedere per livelli di astrazione differenti. Vi sono infatti circostanze in cui non interessa la costituzione intima di un'entita' sintattica a cui ci si riferisce. Ad esempio nella precedente procedura *appendPS* bastava sapere che le liste da concatenare erano costituite da clausole, senza entrare nella loro struttura interna. In altre occasioni invece, e' importante avere accesso alla struttura di un oggetto sintattico per poterlo manipolare adeguatamente. A questo scopo, 'Log offre due meta-rappresentazioni di ogni entita' sintattica in modo da riferirsi ad essa a due livelli di dettaglio diversi. Passare dall'una all'altra meta-rappresentazione e' facile: basta utilizzare l'operatore <=>. Volendo scendere ad un livello di dettaglio maggiore, e' di nuovo disponibile <=>, con cui entrare nella struttura delle entita' viste come atomiche al livello corrente di astrazione.

Una seconda motivazione all'introduzione di una doppia meta-rappresentazione deriva da considerazioni sull'**implementazione** del linguaggio. La rappresentazione strutturale richiede una rappresentazione interna delle diverse entita' sintattiche del linguaggio che e' essenzialmente quella delle liste e comunque tale da potere operare su essa con la normale procedura di unificazione. E' chiaro che se questa fosse l'unica rappresentazione interna si avrebbe un inaccettabile calo dell'efficienza complessiva del linguaggio. Basti pensare all'"overhead" introdotto dalla rappresentazione interna dei simboli come liste di caratteri.

D'altro canto, un'implementazione efficiente del linguaggio richiederebbe piuttosto di adottare rappresentazioni interne ad-hoc per programmi, clausole, termini, ecc. che ne agevolino la gestione. Ad esempio, per i programmi, si potrebbero avere strutture dati ausiliarie (come indici, tabelle hash,...) che facilitino operazioni di ricerca. Avendo pero' quest'ultima come unica rappresentazione interna e volendo altresi' potere accedere alla struttura delle diverse entita' sintattiche (cosi' come avviene con la meta-rappresentazione strutturale), si andrebbe incontro a grossi problemi, dovendo prevedere operazioni ad hoc per i diversi casi in sostituzione della normale unificazione.

La soluzione adottata in 'Log e' quindi di avere due meta-rappresentazioni della stessa entita' sintattica a cui corrispondono due rappresentazioni interne diverse, anche se strettamente correlate (e in buona parte condividenti le stesse strutture dati). La prima meta-rappresentazione, i nomi, in cui non interessa la struttura interna dell'oggetto denotato permette di adottare la rappresentazione interna piu' efficiente. La seconda, la meta-rappresentazione strutturale, invece comporta una rappresentazione interna tipo lista. Il fatto poi che la rappresentazione strutturale descriva la struttura dell'oggetto denotato in termini dei nomi dei suoi componenti permette di sfruttare direttamente la loro rappresentazione interna piu' efficiente, limitando cosi' l'occupazione di memoria. Infine, la presenza dell'operatore <=> permette di costruire una rappresentazione a partire dall'altra (in particolare una rappresentazione a lista da quella piu' efficiente) soltanto quando questo sia veramente necessario.

7. Meta-predicati predefiniti in 'Log

Con lo schema di naming disponibile in 'Log, e' possibile definire direttamente nel linguaggio stesso tutti i vari meta-predicati tipicamente messi a disposizione da Prolog. Questo significa, da una parte, poter dare a questi meta-predicati una semantica logica, dall'altra, poterli ignorare analizzando formalmente il linguaggio. Ovviamente, un interprete concreto di 'Log dovra', per motivi di

efficienza, essere dotato di un'implementazione a piu' basso livello di questi meta-predicati, che comunque rispecchi il piu' possibile la loro definizione dichiarativa data in 'Log.

A titolo esemplificativo, diamo la definizione 'Log dei meta-predicati *var/1* e *atom/1* di Prolog.

```
var(SN)  :- SN <=> [F|SS], isUpper(F).
atom(SN) :- SN <=> [F|SS], isLower(F).

isUpper(X) :- upperAlphabet(U), isIn(X,U).
isLower(X) :- lowerAlphabet(L), isIn(X,L).

upperAlphabet([%A,%B,…,%Z,%_]).
lowerAlphabet([%a,%b,…,%z,%0,…,%9,…]).

isIn(X,[X|R]).
isIn(X,[A|R]) :- isIn(X,R).
```

Altri meta-predicati del Prolog, quali *name/2* e *=../2*, risultano invece del tutto superflui 'Log.

Utilizzando le capacita' meta di 'Log, e' possibile definire in 'Log stesso una procedura di unificazione fra due termini e quindi usarla per definire altri tipici meta-predicati del Prolog, come *call* e *clause*. La procedura di unificazione puo' essere implementata come un predicato **unify(T1,T2,Subs)**, dove *T1* e *T2* sono i term name corrispondenti ai due termini da unificare, mentre *Subs* codifica le sostituzioni (relative alle variabili del livello oggetto) risultanti, rappresentate come lista di coppie *X/t*, ove *X* e' un nome di variabile e *t* un term structure.

Usando il predicato *unify*, risulta relativamente semplice definire in 'Log la **relazione di provabilita'** (a livello oggetto) di un goal in un programma, sotto forma di un nuovo meta-predicato - chiamato *ecall* per continuita' con EnvProlog - in modo analogo al predicato *demo* descritto in [BoK82] e poi ripreso in MetaProlog. *ecall* e' definito nel seguente modo:

ecall(PN,G,Subs,C)

risolve, al livello oggetto, il goal *G* nel programma denotato dal program name *PN*, riportando in *Subs* le sostituzioni alle eventuali variabili oggetto presenti in G. Il quarto argomento, *C*, e' una lista di vincoli della forma x <=> y, con x e y contenenti entrambi variabili, generati durante il processo di risoluzione di *G*. *G* e' espresso come una lista di term structure. *G* puo' pertanto contenere meta-variabili che potranno venire istanziate alla meta-rappresentazione delle opportune entita' sintattiche dal processo di risoluzione. Per agevolare la scrittura di *ecall*, 'Log mette a disposizione anche una notazione sintetica per le liste di term structure composte da almeno due elementi: se $t_0,\ldots,t_n$, con n>1 sono termini, allora si definisce "$t_0,\ldots,t_n$" = ["t_0",…,"t_n"].

Mostriamo ora alcuni esempi elementari d'uso di *ecall/4*. Esempi piu' significativi, relativi alla manipolazione di programmi come dati, si possono trovare in [Ros89] e [Ros90].

```
prog({p(f(b)). p(X):-q(X). q(a).}).

?- prog(P),ecall(P,"p(#X),q(#Y)",S,[]).
     S --> [`X`/"f(b)",`Y`/`a`];
     S --> [`X`/`a`,`Y`/`a`].

?- prog(P),ecall(P,"p(X)",S,C),X=`a`.
     C --> [],
     S --> [].
```

```
?- ecall({p(a).},"P(#X)",S,[]).
     P --> `p`,
     S --> [`X`/`a`].

?- ecall({p(X,Y):-X<=>Y.},"p(N,f(a))",[],C).
     C --> [],
     N --> 'f(a)'.

?- ecall({p(X,Y):-X<=>Y.},"p(#N,#S)",S,C).
     C --> [`X_1`<=>`Y_2`],
     S --> [`N`/`X_1`,`S`/`Y_2`].
```

In modo analogo alla *ecall*, e' possibile definire in 'Log anche gli altri built-in del Prolog, come *clause*, *assert* e *retract* (indicati in 'Log, rispettivamente, con *eclause*, *eassert* ed *eretract*). Ad esempio,

```
?- eclause({p(X,Y):- q(X),r(Y,a).},"p(b,#X):-|B",S).
     B --> ['q(b)','r(Y_1,a)'],
     S --> [`X`/`Y_1`].
```

eassert ed *eretract* possono essere definiti in modo simile ai predicati *add_to* e *drop_from* del MetaProlog descritti in [Bow85] (e poi ulteriormente discussi in [GM*88]). In entrambi questi predicati, e' presente un parametro che specifica il nuovo program name risultante dall'aggiunta o rimozione della clausola data. In particolare, *eassert* richiede che la clausola da aggiungere al programma sia specificata tramite il suo clause name, eliminando cosi' di fatto i problemi relativi alla quantificazione delle eventuali variabili presenti nella clausola (cfr. ad esempio [GM*88]).

8. Note sulla semantica di 'Log

'Log possiede tutte le proprieta' semantiche di un linguaggio di programmazione logica puro come descritte, ad esempio, in [Llo87]. Ci limiteremo pertanto a mettere in rilievo soltanto gli aspetti piu' caratteristici di 'Log. Per maggiori dettagli, si rimanda a [Cer91].

Le maggiori differenze rispetto al caso standard sono dovute alla presenza dell'operatore <=>/2. <=> e' un operatore interpretato. Esso definisce la relazione che deve sussistere tra due espressioni sintattiche. Qualora queste espressioni non siano sufficentemente istanziate, l'intero termine (con <=> come funtore principale) viene restituito come vincolo sui valori assumibili dalle (meta-) variabili che compaiono in esso. L'insieme dei valori associabili delle variabili che compaiono nel codominio della sostituzione calcolata viene di conseguenza limitato. Quest'utilizzo di <=> ha molto in comune con l'approccio del Constraint Logic Programming [JaL87][LiS90].

Verranno ora date alcune definizioni preliminari relative all'operatore <=>. Un termine della forma x<=>y, con x e y termini e <=> come funtore principale e' detto un **R-termine**. Un R-termine x<=>y per il quale esista una sostituzione ground θ tale che $x\theta$ e' un nome e $y\theta$ e' la corrispondente rappresentazione strutturale e' detto **proprio**, altrimenti e' **improprio**. Un R-termine x<=>y in cui sia x che y contengono variabili e' un **R-termine aperto**; se esattamente uno fra x e y contiene delle variabili, si parla di **R-termine semi-ground**; nel caso infine in cui ne' x ne' y contengano variabili, si ha un **R-termine ground**.

Giochera' un ruolo notevole nella successiva discussione l'insieme $\Re$ degli R-termini ground propri o **R-universo**. $\Re$ e' un insieme infinito e ricorsivo: l'appartenenza di un termine ad esso si puo' facilmente dimostrare decidibile. Si noti che il R-universo e' un sottoinsieme dell'universo di Herbrand **H**. Quest'ultimo viene definito nel solito modo, se non per il fatto che la sua costruzione parte da un insieme di caratteri, l'alfabeto, e non dall'insieme dei simboli funzionali e predicativi che compaiono in un programma.

Il concetto di **interpretazione** di un programma 'Log differisce da un'interpretazione di Herbrand tradizionale solamente per la richiesta che il R-universo sia sempre un sottoinsieme dell'insieme dei termini ground validi in essa. Le definizioni di **validita'**, **modello** e **conseguenza logica** vengono date nel modo usuale. E' allora possibile associare ad ogni programma P un **modello minimo** M_P in modo analogo al caso tradizionale. E' inoltre possibile caratterizzare un programma per mezzo di un funzionale T_P definito nel seguente modo: per ogni interpretazione I,

$T_P(I) = \Re \cup \{A \in \mathbf{H} : A\text{:-}A_1,\ldots,A_n$ e' un'istanza ground di una clausola di P e $A_1,\ldots,A_n \in I\}$

dove $\Re$ e' il R-universo. T_P e' ovviamente monotono e si puo' provare che e' pure continuo. Si noti che per ogni interpretazione I, $T_P(I)$ e' infinito. Si dimostra infine che il modello minimo di un programma P coincide con il minimo punto fisso di T_P.

Avendo accennato all'interpretazione dichiarativa di un programma 'Log, passiamo ora alla sua semantica operazionale e ai risultati di correttezza e completezza che le legano.

Un **reificatore** e' una congiunzione finita ed eventualmente vuota $R_1 \wedge \ldots \wedge R_n$ di R-termini aperti e propri per cui esista una sostituzione θ tale che $(R_1 \wedge \ldots \wedge R_n)^\theta$ sia valida in ogni interpretazione (ossia tale che per ogni i, i=1,…,n, $R_i^\theta \in \Re$). Siano P un programma e G = :-$A_1,\ldots,A_n$ un goal 'Log. Una **risposta** a $P \cup \{G\}$ e' una coppia (θ,R), ove θ e' una sostituzione di variabili di G e Q e' un reificatore, tale che nessuna delle variabili che compaiono in R e' presente nel dominio di θ. Una **risposta corretta** a $P \cup \{G\}$ e' una risposta (θ,R) tale che

$$P \models \forall(R \rightarrow (A_1 \wedge \ldots \wedge A_n)^\theta).$$

E' ora possibile dare una caratterizzazione della relazione che intercorre in 'Log tra risposte corrette, validita' e modello minimo.

Teorema 1.
Siano P un programma, G = :-$A_1,\ldots,A_n$ un goal e (θ,R) una risposta per $P \cup \{G\}$ tale che $R \in \Re$ e $(A_1 \wedge \ldots \wedge A_n)^\theta$ sia ground, allora le seguenti affermazioni sono equivalenti:

1. (θ,R) e' una risposta corretta.
2. $(A_1 \wedge \ldots \wedge A_n)^\theta$ e' una conseguenza logica di P.
3. $(A_1 \wedge \ldots \wedge A_n)^\theta$ e' valida in M_P.

Veniamo ora alla definizione di una variante del metodo di SLD-risoluzione adeguata a costituire il motore inferenziale di un interprete 'Log. Siano G = :-$A_1,\ldots,A_m,\ldots,A_n$ un goal e C = B:-$B_1,\ldots,B_q$ una clausola. Il goal G' e' (immediatamente) **derivato** da G e C per mezzo della sostituzione se:

- θ e' un mgu del letterale selezionato A_m e di B (i quali possono eventualmente essere variabili), e G' e' il goal :-$(A_1,\ldots,B_1,\ldots,B_q,\ldots,A_n)^\theta$, oppure

- A_m e' il R-termine proprio ground o semi-ground x<=>y, θ e' il mgu di y e della rappresentazione strutturale corrispondente al nome x, se x e' ground o di x e del nome corrispondente alla rappresentazione strutturale y altrimenti, e G' e' il goal :-(A_1,…,A_{m-1}, A_{m+1},…,A_n)$^\theta$. In questo caso, assumeremo che C sia la clausola vuota.

Siano P un programma e G un goal. Una **derivazione** δ di P$\cup${G} e' una sequenza (finita o infinita) di terne <(G_0,C_0,θ_0),(G_1,C_1,θ_1),…> tale che G_0 e' G e per ogni i>1, G_i e' immediatamente derivata da G_{i-1} per mezzo di C_{i-1} e θ_{i-1}. Una **refutazione** ρ di P$\cup${G} e' una derivazione finita <(G_0,C_0,θ_0),…,(G_n,C_n,θ_n)> tale che G_n sia un reificatore. La **risposta calcolata** da ρ a P$\cup${G} e' la coppia (θ,G_n) ove θ e' la restrizione della composizione θ_1…θ_n alle variabili che compaiono in G.

Questa definizione di refutazione e' piu' generale di quella che viene data per la programmazione logica tradizionale. Una refutazione Prolog e' infatti una refutazione 'Log in cui non viene mai applicato un passo di derivazione utilizzando un R-termine e che ha la clausola vuota come reificatore finale.

Possiamo ora enunciare il teorema di correttezza del metodo di risoluzione adottato in 'Log.

Teorema 2 (Correttezza del metodo di risoluzione).
Siano P un programma e G un goal. Ogni risposta calcolata di P$\cup${G} e' una risposta corretta a P$\cup${G}.

Per provare anche la completezza del metodo di risoluzione di 'Log e' necessario dare prima una nuova definizione. Siano R = $R_1\wedge\ldots\wedge R_n$ e Q = $Q_1\wedge\ldots\wedge Q_m$ due reificatori. R e' **piu' generale** di Q se R impone vincoli meno stringenti di Q sulle variabili che compaiono in entrambi; piu' formalmente, R e' piu' generale di Q se esiste una sostituzione θ tale che per ogni i = 1,…,n, $R_i^\theta \in \Re$ oppure $R_i^\theta = Q_j$ per qualche j = 1,…,m, e per ogni j, j = 1,…,m, se Q_j contiene una variabile presente in R, allora vi e' un i = 1,…,m tale che $R_i^\theta = Q_j$.

Teorema 3 (Completezza del metodo di risoluzione).
Siano P un programma e G un goal. Se (θ,R) e' una risposta corretta a P$\cup${G}, allora esistono una risposta calcolata (σ,Q) di P$\cup${G} e una sostituzione γ tali che $\theta = \sigma\gamma$ e Q e' piu' generale di R.

Come nel caso standard, e' possibile dimostrare che la regola utilizzata per scegliere il letterale selezionato ad ogni passo di refutazione non ha nessuna influenza sulla possibilita' di determinare una refutazione, qualora esista. In questo senso, vale un teorema di completezza forte con cui si possono imporre vincoli arbitrari sulla modalita' di selezione dei letterali da risolvere.

9. Confronto con altre proposte

Il linguaggio 'Log e' basato sull'idea di **amalgamare** linguaggio oggetto e meta-linguaggio nel contesto di un linguaggio logico, inizialmente proposta da Bowen e Kowalski in [BoK82]. In quest'ottica, 'Log presenta molte affinita' con **MetaProlog**, un linguaggio di programmazione logica sviluppato da Bowen e altri alla Syracuse University [Bow85][BoW85]. Seguendo le idee originarie di Bowen e Kowalski, MetaProlog sostituisce alcuni meta-predicati impuri di Prolog

(quali *assert* e *retract*) con le nozioni meglio fondate di teoria, dimostrazione in una teoria e creazione dinamica di nuove teorie.

Contrariamente a 'Log, MetaProlog definisce veri e propri nomi per le **sole teorie**, mentre non sviluppa adeguatamente una relazione di naming per altre entita' sintattiche. Inoltre, 'Log utilizza nomi strutturalmente descrittivi, mentre in MetaProlog la meta-rappresentazione di una teoria e' una **semplice costante**, priva di alcuna rassomiglianza strutturale con l'oggetto da essa denotato. Di conseguenza, come si puo' leggere in [BoW85], "All MetaProlog program databases … are set up either by reading them in from a file or by dynamically constructing them using system predicates". In 'Log, invece, un program name elenca le clausole che lo compongono. La soluzione 'Log risulta pertanto adeguata alla **modularizzazione** dei programmi: un'intero programma puo' infatti essere partizionato in moduli (utilizzando program name o program structure), possibilmente annidati, accessibili solamente mediante meta-predicati quali *ecall* (cfr. [Ros90]).

'Log permette inoltre la presenza di meta-variabili nei program structure, mentre questa possibilita' non e' disponibile in MetaProlog. Una conseguenza di cio' e' l'impossibilita' di definire in MetaProlog meta-predicati quali *appendPS*, definito nel capitolo 4 (se non simulandolo con una serie di *add_to*). La presenza di nomi strutturalmente descrittivi rende tuttavia 'Log inadeguato alla costruzione di frasi auto-referenziali. Infine, una caratteristica di MetaProlog che merita di essere menzionata e' che esistono varie implementazioni efficienti di questo linguaggio descritte nella letteratura [Bac86][Cic88].

Uno schema di naming piu' completo e' stato proposto da **Barklund** [Bar88]. Gli obiettivi di questo autore sono molto vicini ai nostri: definire "... a naming of Prolog formulas and terms as Prolog terms to create a practical and logically appealing language for reasoning about terms, programs,…". 'Log condivide inoltre con la proposta di Barklund l'utilizzo di nomi strutturalmente descrittivi, la nozione di meta-variabile e la definizione di meta-programma (in particolare, il fatto che non si deve tentare di classificare i programmi in meta-livelli). La maggiore differenza con la proposta di quest'autore deriva dalla presenza di una doppia meta-rappresentazione in 'Log. Cio' diversifica 'Log dalla soluzione di Barklund sia per le sue modalita' d'uso che per la sua implementazione. Ad esempio, la clausola

```
p(X,f(a)) :- q(X), r(a,b)
```

e' rappresentata nella proposta di Barklund dal termine

```
clause(atom(p,[var(0),compound(f,[const(a)])]),
       conj(atom(q,[var(0)]),atom(r,[const(a),const(b)])))
```

mentre in 'Log la sua rappresentazione strutturale (in notazione esplicita) e'

```
clause('p(X,f(a))',['q(X)','r(a,b)']).
```

E' poi possibile, in 'Log, accedere alla struttura dei termini costituenti questa clausola, per mezzo dell'operatore <=>, solamente se necessario. Questa caratteristica differenzia 'Log da numerose altre proposte che si rifanno ad una meta-rappresentazione strutturale simile a quella descritta da Barklund (ad esempio Reflective Prolog [CoL89]). Queste proposte non offrono uno schema di naming applicabile a **tutte** le entita' sintattiche di un linguaggio logico. Le due classi sintattiche estreme – programmi e simboli – sono infatti generalmente escluse. Infatti, disponendo solamente delle meta-rappresentazioni strutturali, l'utilizzo di nomi di programmi (e clausole) al meta-livello tende a diventare molto pesante per l'**utente**, mentre il mantenimento della rappresentazione strutturale delle entita' di basso livello, quali i simboli, puo' risultare estremamente costoso per l'**interprete**. Come

gia' fatto notare nel capitolo 6, la presenza di due meta-rappresentazioni distinte permette di superare entrambi questi problemi.

Un'altra proposta molto interessante e' il linguaggio **Gödel**, recentemente definito da J.Lloyd e altri alla Bristol University [BHL90]. Questo linguaggio ha esattamente gli stessi obiettivi di 'Log: "The main design aim of Gödel is to have functionality and expressiveness similar to Prolog, but to have greatly improved declarative semantics compared with Prolog". Gödel e' tuttavia caratterizzato da un complesso **sistema di tipi** che influenza pesantemente la struttura del linguaggio, rendendo altresi' difficile un confronto effettivo con 'Log. Cosi' come 'Log, Gödel offre due meta-rappresentazioni di ogni entita' sintattica, chiamate rappresentazione *ground* e *non-ground* rispettivamente. Queste due meta-rappresentazione si differenziano principalmente per il modo in cui rappresentano le variabili oggetto: mediante meta-variabili nella prima, per mezzo di meta-termini ground nella seconda. Le ragioni che giustificano la scelta di una doppia meta-rappresentazione sono tuttavia lontane dai motivi addotti per 'Log riportati nel capitolo 6. In particolare, in Gödel il doppio schema di naming non ha lo scopo di descrivere le entita' rappresentate a due livelli di dettaglio diversi. Inoltre, le due meta-rappresentazioni di questo linguaggio non sono "cooperanti". Non e' ad esempio possibile passare dinamicamente dall'una all'altra come in 'Log. [BHL90] menziona anche alcuni problemi implementativi di Gödel: "The most important of these is the implementation of the ground representation". Questi problemi (almeno quelli menzionati in [BHL90]) non sembrano sorgere in 'Log.

Vi sono infine altre proposte aventi vari punti in comune con 'Log, ma che se ne differenziano per obiettivi e approccio progettuale. Fra queste, vi e' Reflective Prolog, in cui, differentemente da 'Log, il metodo di risoluzione viene esteso con un meccanismo di "implicit reflection" [CoL89]. Dunque, "Reflective Prolog is not an amalgamated language in the sense of [BoW82]", dal momento che la risoluzione cosi' estesa permette di dimostrare nuovi teoremi non dimostrabili nel linguaggio stesso.

10. Conclusioni

In questo lavoro, sono state presentate le capacita' di meta-programmazione disponibili nel linguaggio 'Log, discutendone gli aspetti innovativi, i possibili usi e la semantica. E' stato mostrato come la presenza di un doppio schema di meta-rappresentazione opportunamente scelto e di un unico operatore (<=>) sono una base sufficiente a definire in modo dichiarativo ogni altro strumento di meta-programmazione di Prolog.

E' in corso di realizzazione un'implementazione di 'Log che prevede l'estensione di un interprete Prolog scritto in Modula 2 con i costrutti di meta livello definiti nei capitoli precedenti. Finora sono stati implementati con successo i program name, i program structure e l'operatore <=> ristretto a queste entita' (per altro gia' implementati per il linguaggio EnvProlog [Ros90] anche se in modo piu' approssimativo). In un prossimo futuro, verranno implementate anche le altre meta-rappresentazioni previste da 'Log seguendo la stessa tecnica adottata per la rappresentazione dei programmi.

Riferimenti bibliografici

[ACS86] Aiello L., Cecchi C., Sartini D.: "Representation and Use of Metaknowledge" in *Proc. of the IEEE*, vol 74, n. 10, Oct. 1986, pp 1304-1321.

[Bac86] Bacha, H: "Meta-level Programming: A Compiled Approach", *4th Int. Conf. on Logic Programming*, Melbourne, 1986.

[Bar88] Barklund J.: "What is a Meta-variable in Prolog?", in *Workshop on Meta-Programming in Logic Programming* (Lloyd J.W. ed.), 1988.

[Bow85] Bowen K.A.: "Meta-Level Programming and Knowledge Representation" *New Generation Computing* 3 (1985), pp 359-383.

[BoK82] Bowen K.A., Kowalski R.A.: "Amalgamating Language and Metalanguage in Logic Programming", in *Logic Programming* (K.L. Clark and S.A. Tarnlund, Eds.), Academic Press, 1982, pp 153-172.

[BoW85] Bowen K.A., Weinberg T.: "A Meta-Level Extension to Prolog" in *IEEE Symposium on Logic Programming*, Boston 1985, pp 669-675.

[BHL90] Burt A.D., Hill P.M., Lloyd J.W.: "Preliminary Report on the Logic Programming Language Gödel", Int. Rpt. TR-90-02, Univ. of Bristol, Dept. of Computer Science, 1990.

[Cer91] Cervesato I.: "Una proposta per l'introduzione di capacita' di meta-rappresentazione in un linguaggio di programmazione logica" (in italian), Tesi di Laurea in Scienze dell'informazione, Universita' di Udine, 1991.

[Cic88] Cicekli, I.: "Design and Implementation of An Abstract MetaProlog", in *Workshop on Meta-Programming in Logic Programming* (Lloyd J.W. ed.), 1988, pp 307-318.

[CoL89] Costantini S., Lanzarone G.A.: "A Metalogic Programming Language.", in *Logic Programming, Proceedings of the Sixth International Conference* (G. Levi, M. Martelli, Eds), MIT Press, 1989.

[GM*88] Giordano L., Martelli A., Murgia I., Rossi G.: "Alcune Considerazioni sulla Integrazione tra Livello Oggetto e Meta nel Linguaggio Logico MI_Prolog", *Terzo Convegno Nazionale sulla Programmazione Logica (Gulp 88)*, 1988, pp 131-148.

[JaL87] Jaffar J., Lassez J.L.: "Constraint Logic Programming" in *14th POPL*, Munich, West-Germany, 1987.

[LiS90] Lim P., Stuckey P.J.: "Meta-Programming as Constraint Programming", in *Proceedings of the 1990 North American conference on Logic Programming* (Debray S., Hermenegildo M., Eds.), 1990, pp 416-430.

[Llo87] Lloyd J.W.: "*Foundations of Logic Programming*", Springer-Verlag, II ed., 1987.

[MaR88] Martelli A., Rossi G.F.: "Enhancing Prolog to Support Prolog Programming Environments", in *ESOP88 - L.N.C.S. n.300* (H.Ganzinger, Ed.), Springer-Verlag, 1988.

[Ros89] Rossi G.F: "Meta-programming Facilities in an Extended Prolog", in *Artificial Intelligence and Information-Control Systems for Robots* (I.Plander, Ed.), North-Holland, 1989

[Ros90] Rossi G.F: "Programs as Data in an Extended Prolog", Research Report n. RR-07-90, Dipartimento di Matematica e Informatica, University of Udine, 1990.

[SaS86] Safra S., Shapiro E.: "Meta-interpreters for Real", in *Information Processing 86* (H.-J. Kugler Ed.). North-Holland, 1986

[StL88] Sterling L., Lakhotia A.: "Composing Prolog Meta-interpreters", *Fifth Int. Conf. on Logic Programming*, Seattle, 1988.

[StS86] Sterling L.S., Shapiro E.: "*The Art of Prolog*", MIT Press, 1986.

[Sug90] Sugano H.: "Meta and Reflective Computation on Logic Programming and Its Semantics", in *Proc. 2nd Workshop on Meta-Programming in Logic Programming* (Bruynooghe M., Ed.), April 1990, pp 19-34.