optimization. Proc. ECAI-84: Advances in Artificial Intelligence (T. O'Shea, Ed.), North Holland, 1984, 91-100.



# UN'ESTENSIONE DELL'INTERPRETE PROLOG CON MODULARITA' E CONCORRENZA

*Paola Mello* (*), *Antonio Natali* (*),
*Aurelio Rossolini* (*), *Francesco Russo* (**)

(*) DEIS - Facolta' di Ingegneria - Bologna
(**) ENIDATA - SOPEB - Bologna

## Sommario

In questo lavoro si descrivono le estensioni apportate ad un interprete Prolog per consentire la definizione di programmi Prolog modulari e concorrenti in ambiente mono-processore. In particolare viene introdotto un insieme di primitive che consentono l' uso del Prolog come linguaggio di sistema. In questo modo la potenza espressiva del linguaggio e' sfruttata per la prototipazione di diversi "nuclei" di sistema e differenti ambienti di programmazione eliminando la necessita' di ricorrere ad altri linguaggi o strumenti. Le estensioni descritte sono state utilizzate quale supporto implementativo del modello CPU (Communicating Prolog Units), precedentemento proposto per la definizione di modularita' e concorrenza in Prolog.

Il lavoro e' stato parzialmente finanziato da ENIDATA S.p.A., CEE (Progetto Esprit P973) e MPI (40%).

## INTRODUZIONE

Due concetti sembrano essere fondamentali per l'utilizzo del linguaggio Prolog quale linguaggio di sistema per la rapida prototipazione di differenti supporti a tempo di esecuzione e per la costruzione di ambienti di programmazione integrati: la modularita' e la concorrenza. Lo scopo principale del lavoro qui descritto e' stato quello di individuare ed implementare i meccanismi base che devono essere inseriti nel linguaggio Prolog per consentire tale possibilita', naturalmente mantenendo con Prolog piena compatibilita'.

L' obiettivo e' di fornire un insieme minimale di meccanismi tale da consentire di implementare in Prolog, in modo altamente flessibile, differenti politiche di combinazione fra moduli e di comunicazione e sincronizzazione fra processi.

Per quello che riguarda la modularita', soprattutto analizzando proposte che tendono ad estendere Prolog con concetti tipici della programmazione ad oggetti ([1], [2]), si e' arrivati alla conclusione che il meccanismo con cui costruire

forme piu' sofisticate di modularita', era la possibilita' di partizionare il singolo database Prolog in un insieme di database fisicamente separati e comunicanti mediante la richiesta di goal da dimostrare. Partendo da questo meccanismo, forme di protezione, import/export fra moduli, ereditarieta' possono essere implementate direttamente in Prolog.

Relativamente alla concorrenza, la ricerca di uno strumento particolarmente flessibile ha tratto ispirazione dal linguaggio Modula-2 [3], che consente la definizione esplicita di nuclei per il supporto di processi concorrenti in sistemi monoprocessore. Le estensioni al linguaggio Prolog risultanti non presuppongono predefinite politiche di sincronizzazione e schedulamento, ma si limitano a fornire strumenti per specificare il cambio di contesto fra diversi interpreti Prolog "quasi concorrenti". Cio' consente di lasciare al programmatore la possibilita' di definizione degli strumenti di comunicazione e sincronizzazione nonche' le strategie di schedulamento piu' adatte alla specifica applicazione.

Nell'implementazione dei meccanismi cosi' individuati si sono voluti rispettare i seguenti vincoli:

1- Ampia portabilita' e flessibilita'. Per questo motivo non si e' voluto fare riferimento diretto alle politiche di creazione e gestione dei processi di un particolare Sistema Operativo (ad esempio UNIX).

2- Buona efficienza. Per questo motivo non era accettabile implementare le primitive specificate con la tecnica della meta-interpretazione [4]. Anche applicando tecniche di valutazione parziale [5] le prestazioni del sistema erano troppo basse, considerando che su tale sistema si dovevano poi poter costruire politiche, anche complesse, di interazione fra unita' separate e processi.

La scelta adottata, che consente un buon compromesso fra le esigenze di portabilita' ed efficienza, e' stata quella di implementare in linguaggio C le estensioni necessarie, modificando un interprete C-Prolog in ambiente UNIX BSD 4.2.

L'interprete Prolog cosi' esteso (Prolog+) e' stato utilizzato, con risultati soddisfacenti, per l'implementazione del modello Communicating Prolog Units [6] ed e' in fase di studio il suo impiego nel contesto del progetto Esprit P973, quale nucleo per la costruzione di un ambiente avanzato per la programmazione logica.

L'articolo e' organizzato come segue. Nel primo paragrafo si presentera', molto brevemente, l'organizzazione dell'interprete C-Prolog. Nel secondo paragrafo si discuteranno i meccanismi base introdotti per consentire la modularita' e la loro implementazione. Nel terzo paragrafo si delineeranno le primitive che consentono di costruire una macchina virtuale Prolog a processi e le modifiche all'interprete C-Prolog introdotte. Nel quarto paragrafo si mostrera' qualche semplice esempio di utilizzo di Prolog+ sottolineandone le caratteristiche essenziali.



## 1. L'ORGANIZZAZIONE DELL'INTERPRETE C-PROLOG

La presentazione dell'organizzazione e del funzionamento dell'interprete C-Prolog vuol essere solo introduttiva alla descrizione delle modifiche ad esso apportate. Per un' analisi piu' approfondita si consulti [7].

Con l'intento di fornire all'utente una macchina virtuale Prolog efficiente e nello stesso tempo flessibile, l'interprete C-Prolog e' stato strutturato in due livelli con caratteristiche diverse. Il livello piu' basso, scritto in linguaggio C, realizza il "motore inferenziale" dell'inteprete ovvero implementa l'algoritmo di unificazione e di navigazione del proof_tree del Prolog. E' a questo livello che vengono risolti i problemi di interfaccia con il sistema operativo ed e' a questo livello che bisogna intervenire per modificare il comportamento del motore inferenziale, ad esempio per cambiare la regola di computazione o la regola di ricerca.

Il livello piu' alto dell'interprete (supervisore), scritto in linguaggio Prolog, realizza gli strumenti di controllo e di interfaccia con l'utente sfruttando la capacita' del livello sottostante. In particolare al momento della sua attivazione l'interprete consulta alcuni file contenenti le definizioni dei predicati di sistema e mette in esecuzione un goal particolare chiamato 'live'. La definizione del predicato 'live', fornito per esteso in appendice A, costituisce il loop di controllo ad alto livello dell'interprete.

L'interfaccia fra l'interprete scritto in C e il livello di supervisore avviene mediante dei predicati predefiniti che hanno una forma del tipo:

nome_pred( <argomenti> ) :- <numero_intero>.

Alla chiamata di tali predicati, verra' eseguita la parte di codice C associata al numero intero indicato. Predicati di questo tipo sono deterministici e quindi falliscono in caso di backtracking. Questo tipo di meccanismo permette di estendere e personalizzare l'interprete in modo agevole. Per aggiungere un nuovo predicato predefinito e' sufficiente scriverne il codice C corrispondente e associarlo alla definizione Prolog secondo il meccanismo descritto.

C_Prolog usa per la rappresentazione delle clausole la tecnica detta Structure Sharing [7]. In particolare, il puro codice e' rappresentato attraverso un insieme di strutture dati particolari (atom, functor, skel, clauserec) tra loro strettamente connesse, come vedremo meglio nel prossimo paragrafo. Dette strutture dati sono inserite in due aree di lavoro, chiamate Atom Space e Heap Space, di dimensione prestabilita e sono nel loro insieme la descrizione della base di conoscenza dell'interprete.

Piu' in dettaglio, l'Atom Space e' utilizzato per contenere informazioni riguardanti gli atomi Prolog ovvero la loro rappresentazione sotto forma di stringa, il tipo di operatore e le precedenze per gli atomi usati come operatori ecc. L' Heap Space contiene informazioni sui funtori, le clausole e quindi in generale la descrizione dei termini composti.

Per quanto riguarda i binding environment e la descrizione del proof_tree,

C_Prolog utilizza altre quattro aree di lavoro di dimensione prestabilita, chiamate Local, Global, Trail e Auxiliary gestendole a stack. In particolare nel Local stack sono inserite le strutture dati chiamate Local Frame contenenti, per ogni passo di risoluzione, le informazioni relative al nodo corrente del proof_tree e i binding environment delle variabili locali; il Global Stack e' sfruttato per i binding environment delle variabili globali; il Trail Stack e l'Auxiliary stack sono utilizzati rispettivamente per contenere informazioni necessarie in caso di backtracking e durante operazioni di input/output.

Il motore inferenziale, infine, per poter gestire l'algoritmo di navigazione del proof_tree utilizza una serie di variabili che possono essere considerate come dei veri e propri registri dedicati della macchina virtuale Prolog.

## 2. MODULARITA'

### 2.1 Il meccanismo base

Se si segue l'interpretazione procedurale di Prolog, il modo piu' semplice e naturale di considerare un modulo Prolog (denominato UNITA' nel seguito) e' come un insieme di clausole. Tale concetto di modularita', adottato da quasi tutte le proposte che estendono Prolog col concetto di modularita' [8] o di oggetto ([7],[2]), e' fondamentale per la costruzione di programmi Prolog "in-the-large" e per la strutturazione della conoscenza in gerarchie [9] o contesti [1].

Quindi un'unita' e' un insieme di clausole Prolog identificata da un nome unico nel sistema. Goals eseguiti in unita' diverse possono dare risultati diversi. Ogni unita' e' un mondo aperto [11] capace di richiedere la dimostrazione di goals ad altre unita'. Per specificare l'unita' in cui si vuole tentare una dimostrazione per il goal G, e' stato introdotto un nuovo predicato di sistema:

**ask(Unit,G).**

Supponiamo che ask(Unit,G) sia invocato nel contesto dell'unita' u1. Allora, il goal G e' risolto nel contesto dell'unita' Unit. Se G ha successo, ask(Unit,G) e' considerato risolto nel contesto di u1. In caso di successivo backtracking su 'ask' si esplora un'altra alternativa di G. Se G fallisce o se Unit non esiste, anche 'ask' e' considerato fallito e in u1 e' attivato il backtracking.

Il predicato 'ask' puo' essere considerato un operatore non-monotono che consente di modificare dinamicamente l'insieme delle clausole da utilizzare per la dimostrazione di un goal. Molti sistemi basati sulla logica, e su Prolog in particolare, utilizzano altre forme di combinazione fra unita' quali operatori di crescita dinamica di contesto ([1]), differenti forme di ereditarieta' ([12]) ecc. La scelta e' caduta sul predicato predefinito di 'ask' in quanto, sulla sua base, possono essere costruite le forme piu' sofisticate di composizione di unita' precedentemente citate.

Per mantenere con Prolog piena compatibilita', i predicati di sistema devono essere visibili in ogni unita'. Per questo motivo, le loro definizioni sono, concettualmente, inserite in ogni unita'.



### 2.2 L'implementazione

L'associazione di un insieme di clausole ad una unita' non richiede logicamente alcun meccanismo particolare a livello di macchina virtuale. Tuttavia, seguendo il criterio dei "piccoli domini di accesso" tipico dei sistemi operativi e orientati agli oggetti, Prolog+ supporta una separazione "fisica" delle diverse unita'. Per ottenere tale separazione, e' necessario intervenire sulla modalita' di rappresentazione interna delle clausole del C_Prolog.

Nell'interprete, il puro codice di ogni clausola e' rappresentato da un record descrittore (CLAUSEREC): in fase di consultazione i record CLAUSEREC sono suddivisi in liste separate in base al funtore principale della testa della clausola che descrivono. Per separare le clausole che appartenegono ad unita' diverse, occorre ulteriormente suddividere tali liste.

Come rappresentato in figura 1, il puntatore alla lista di clausole con stessa testa viene sostituito con un puntatore ad una lista di descrittori di unita'. Ogni descrittore contiene il nome dell' unita' ad esso associata (sotto forma di atomo), il puntatore alla lista delle clausole in essa contenute ed un puntatore al successivo descrittore.

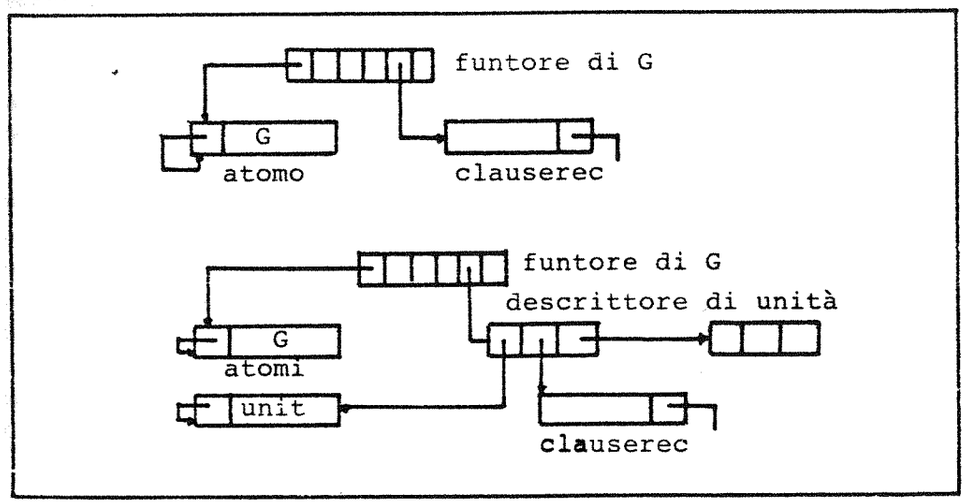


**Figura 1**

E' stato quindi introdotto un nuovo registro nella macchina virtuale Prolog (**Unit-reg**) con il compito di mantenere il nome dell' unita' corrente, cioe' dell'unita' di cui devono essere considerate le clausole per la dimostrazione del goal corrente. Il motore inferenziale, ad ogni passo di risoluzione, lavora sempre solo con le clausole inserite nella unita' indicata da Unit-reg. Sempre lo stesso registro guida le operazioni di interazione con il data base (consult,

reconsult, assert, retract, clause); risultato di tali operazioni e' la modifica delle sole strutture dati relative alla unita' referenziata da Unit-reg.

La scelta di organizzare i descrittori di unita' in una lista anziche' in un array permette di non fissare staticamente il numero massimo di unita' definibili anche se costringe ad una scansione sequenziale della lista per ogni goal da dimostrare. Il degrado delle prestazioni nella dimostrazione di un goal G in una unita' e' proporzionale al numero di unita' che presentano clausole con lo stesso funtore principale e numero di argomenti di G. In appendice B sono riportati alcuni risultati di prove di performance. D'altro canto la scelta di un array implicherebbe un'occupazione di memoria molto superiore, proporzionale alla dimensione dell'array per il numero di funtori. Una diversa organizzazione,attualmente allo studio, subordina l'accesso al funtore all'accesso al descrittore di unita'.

Per quello che riguarda la visibilita' dei predicati di sistema possono essere considerati tre tipi diversi di soluzione:

1. tutti i predicati di sistema sono replicati in ogni unita';
2. tutti i predicati di sistema sono contenuti in una particolare unita' 'system' alla quale far riferimento con 'ask/2';
3. tutti i predicati di sistema sono inseriti in un'unita' globale e quindi implicitamente visibili da tutte le altre unita'.

Fra le tre alternative e' stata scelta l'ultima. La prima, infatti, presenta un' inutile duplicazione di informazioni mentre la seconda appesantisce l'elaborazione con cambi di contesto troppo frequenti e non garantisce compatibilita' col Prolog standard. L'introduzione di un'unita' globale, invece, garantisce una buona efficienza e mantiene piena compatibilita' con il Prolog. Piu' precisamente, quando non trova alcuna definizione per un goal nell' unita' corrente, l'interprete richiede la dimostrazione di quel goal alla unita' globale, individuata dall'atomo nullo (cioe' dal simbolo '[ ]'). Tale situazione e' sempre verificata per i predicati di sistema essendo non ridefinibili dall'utente. L'unita' globale puo' essere anche sfruttata dal programmatore per contenere predicati ritenuti di utilita' generale e renderli, quindi, automaticamente visibili da ogni altra unita'. Chiaramente l'eventuale ridefinizione in un'altra unita' di tali predicati nasconde all' interno di essa la loro precedente definizione nell'unita' globale.

### 2.3 Il cambio di contesto fra unita'

La realizzazione della primitiva ask(U,G) e' simile a call(G) con l'unica differenza che ask(U,G) inserisce il valore di U nel registro Unit-reg prima di iniziare la dimostrazione di G. Il precedente valore del registro Unit-reg viene inserito, nel 'frame' di lavoro assieme alle altre informazioni sul corrente nodo del proof_tree, all'interno del Local Stack dell'interprete. Esso viene automaticamente ripristinato nel caso in cui il procedimento di backtracking si propaghi dall'unita' U all'unita' chiamante.



## 3. CONCORRENZA

### 3.1 I meccanismi base

Il problema dell'attivazione e gestione di attivita' parallele nell'ambito della programmazione logica e' uno dei campi di ricerca tuttora aperti e nel quale confluiscono proposte dalle caratteristiche piu' diverse ([13],[14],[15]). L'obiettivo di questo lavoro e' ben delimitato: consentire la definizione di processi intesi come attivazioni dell'interprete Prolog autonome e concorrenti, mantenendo quindi con Prolog piena compatibilita'. Cosi' definiti i processi non hanno alcun legame di gerarchia fra loro come invece accade in Concurrent Prolog [13], Delta-Prolog [14] o Parlog [15]. La conoscenza "fisicamente" condivisa fra i processi e' solo lo spazio delle costanti e il database della unita' nel contesto della quale sono creati.

Il criterio progettuale che ha ispirato le scelte di Prolog+ e' basato sull'obiettivo di fornire un supporto estendibile e flessibile per la gestione dei processi. Sono stati quindi individuati meccanismi primitivi tali da permettere, da parte del programmatore, la gestione diretta delle attivita' concorrenti secondo modalita' decise in base all'applicazione considerata. Per questo motivo non si sono volute sfruttare direttamente le caratteristiche di multitasking di un sistema operativo (ad esempio UNIX) e quindi i suoi meccanismi di creazione, terminazione, comunicazione e scheduling dei processi, soluzione peraltro perseguita da altri gruppi di ricerca [14]. Tale soluzione, infatti, oltre a limitata portabilita' , introduceva una notevole rigidita' dovuta alla impossibilita' da parte del programmatore di definire politiche diverse di scheduling e comunicazione fra processi.

Si e' dunque definita una "macchina virtuale" secondo criteri simili a quelli di Modula-2 [3] fondata sul concetto di "coroutine". Il predicato:

**activate(Goal,Id).**

crea un processo (coroutine) con goal corrente 'Goal' e con identificatore unico 'Id'. Tale processo opera nel contesto della unita' cui e' stata richiesta la esecuzione di 'activate'.

Un processo termina quando e' terminata, con successo o fallimento, la dimostrazione del goal con cui era stato attivato. Il predicato:

**kill(Id).**

provvede a rilasciare l'area di lavoro e a rendere di nuovo disponibile l'identificatore del processo.

Il trasferimento volontario del controllo dal processo in esecuzione ad un altro processo attivo (Proc_id) e' ottenuto mediante il predicato predefinito:

**transfer(Proc_id).**

Esso fallisce se 'Proc_id' non e' un processo precedentemente attivato. Questo predicato ha caratteristiche simili al comando transfer di Modula-2, ma a differenza di quello ha un solo argomento in quanto i processi sono individuati

da nomi espliciti e non da descrittori referenziati implicitamente da puntatori. Usando la primitiva transfer/1 e' possibile scrivere, in modo molto semplice (si veda il paragrafo 4), uno scheduler che sospenda e riattivi processi secondo politiche definite dall'utente, nonche' esprimere politiche di comunicazione e sincronizzazione fra processi.

Tutti questi predicati sono deterministici e quindi falliscono in caso di backtracking, senza pero' ripristinare la situazione precedente la loro chiamata.

## 3.2 L'implementazione

### 3.2.1 Il contesto di un processo

Il contesto di un processo e' costituito dai valori delle variabili dell'interprete che fungono da registri virtuali piu' la sua area di lavoro gestita a stack (cioe' Local, Global, Trail, Auxiliary).

Le variabili dell'interprete, considerate come registri, sono:

- i puntatori alla struttura dati del goal e della clausola corrente;
- il puntatore al prossimo goal da dimostrare;
- i puntatori agli stack Local, Global, Trail, Auxiliary;
- il registro dell'unita' corrente;
- i puntatori ai frame locali e globali del goal corrente e del goal precedente;
- i puntatori al frame locale e globale dell'ultimo punto di scelta a cui tornare in caso di backtracking;
- i descrittori dei file correnti di input/output;
- i descrittori di alcuni stati particolari dell'interprete (debug o trace mode, il livello di break ecc);
- i valori, infine, degli indirizzi di inizio e fine degli stack di lavoro.

Sono invece considerate condivise, e, quindi, non appartenenti al contesto di un processo, le aree Atom ed Heap in quanto descrivono i database Prolog accessibili a tutti i processi.

### 3.2.2 La creazione di un processo

Per eseguire con successo la primitiva activate(Goal,Id) si devono eseguire le seguenti operazioni:

- Salvataggio del contesto del processo correntemente in esecuzione. A tale scopo e' stato introdotto un nuovo registro macchina (**Proc-reg**) che contiene l'identificatore del processo correntemente in esecuzione. Il salvataggio del contesto avviene sempre in cima al Local stack; il valore del Local stack pointer alla fine di questa operazione viene salvato in un descrittore gestito a livello C e utilizzato in fase di ripristino del contesto.
- Allocazione di una nuova area di lavoro per il nuovo processo. Per rendere piu' agevole questa operazione si e' provveduto a cambiare l'ordine relativo delle sei aree di memoria dell'interprete per separare nettamente l'area di lavoro privata di ogni processo da quella comune a tutti.
- Inizializzazione dei registri virtuali del nuovo processo.
- Esecuzione del goal specificato 'Goal'.



Per quello che riguarda l'ultima operazione, non si puo' solamente forzare il predicato di controllo top_level "live", dal quale sarebbe poi impossibile risalire a 'Goal', ne' si puo' semplicemente forzare l'esecuzione di 'Goal' in quanto un suo eventuale fallimento non consentirebbe di rientrare nel loop di controllo dell'interprete. Per queste ragioni e' stato definito un nuovo predicato:

```
live1 :- call1,success-termination,live.
live1 :- failure-termination.
live1 :- live.
```

Quando un nuovo processo e' attivato, il registro puntatore al goal corrente e' forzato al goal live1. Il puntatore al goal indicato in fase di attivazione, salvato in una variabile C ausiliaria, viene sfruttato dal predicato call1 per forzare l'esecuzione di 'Goal'.

I goal 'success-termination' e 'failure-termination' hanno, per default, valore vero e falso rispettivamente. Essi possono essere ridefiniti dall'utente per implementare particolari politiche di gestione della terminazione. Cio' e' necessario se si desidera evitare che, alla terminazione di un processo, l'interprete ritorni al loop di controllo.

La primitiva 'activate' cosi' implementata ha alcune limitazioni:

1- L'identificatore del processo (Id) deve essere un intero.
2- 'Goal' deve essere atomico.

Queste limitazioni, pero', non riducono il potere espressivo. Ad esempio, se e' necessario passare variabili istanziate come parametri di 'Goal' e' sufficiente asserire nella unita' corrente il goal iniziale, attivare il processo nella stessa unita', ritrarre il goal iniziale e, infine, metterlo in esecuzione.

```
startprocess(Goal,Newp) :-   assert(tostart(Newp,Goal)),
                             activate(livp,Newp).

livp:-   curr_process(NewP),
         retract(tostart(Newp,Goal)),
         Goal.
```

Si noti che le variabili del goal 'Goal' possono essere solo valori di input; eventuali valori di output non sono direttamente visibili nell'ambiente di creazione. Il predicato 'curr_process/1' consente l'esplorazione del registro Proc-reg.

### 3.2.3 La terminazione di un processo

I processi terminano quando la dimostrazione del loro goal iniziale termina con successo o fallimento. Se il programmatore non esprime una particolare politica di terminazione nel codice del processo oppure nei predicati 'success-

termination', 'failure-termination' o 'livp', la politica di default e' quella del top_level Prolog. In questo caso il sistema ripropone il prompt e attende un altro goal da dimostrare. Per consentire, alla terminazione, il rilascio dell'area di memoria occupata dal processo occorre far uso della primitiva 'kill'.

### 3.2.4 Il cambio di contesto tra processi

Il passaggio esplicito del controllo del processore da un processo ad un altro e' ottenuto mediante il predicato 'transfer(Proc_id)'. Tale predicato esegue le seguenti operazioni:

1- Salvataggio del contesto del processo in esecuzione con modalita' analoghe a quelle descritte per il predicate 'activate';
2- Ripristino del contesto del processo individuato dal valore di 'Proc_id';
3- Continuazione dell'elaborazione, nel nuovo contesto, dal punto in cui si era interrotta.

Il cambiamento di contesto puo' anche essere indotto in modo asincrono rispetto alla attivita' del processo corrente, mediante interruzioni. Le interruzioni sono avvertite al termine di ogni passo di risoluzione, quindi alla fine di ogni ciclo dell'interprete, assimilando il passo di risoluzione di una "macchina Prolog" al ciclo di fetch/execute di una macchina di Von Neumann.

La gestione delle interruzioni e' stata per il momento basata sul meccanismo di segnalazione del sistema operativo UNIX. All'arrivo di un segnale viene settato un flag specifico il cui stato viene verificato ad ogni passo di risoluzione. Se si riscontra la presenza di una richiesta di interruzione, lo stato del processo corrente viene salvato e viene forzata l'esecuzione di uno specifico predicato Prolog che funge da routine di risposta all'interruzione. Al termine di tale routine il processo interrotto riprende l'elaborazione a meno di altre politiche di scheduling definite dall'utente. E' in fase di definizione un insieme di predicati che permettano la gestione completa delle interruzioni, ma, seguendo lo schema tracciato, si e' potuto gia' implementare un semplice meccanismo di time-sharing fra i processi che verra' descritto nel paragrafo 4.

## 4. ALCUNI ESEMPI

Le estensioni all'interprete Prolog presentate precedentemente sono state utilizzate per l'implementazione del modello CPU ([6], [16]). Nel seguito si riportano alcune parti di tale implementazione allo scopo di mostrare la flessibilita' dei meccanismi base introdotti.

### 4.1 Uno scheduler in Prolog

In questa versione dello scheduler si presuppone che non esistano interruzioni: per questo motivo non e' necessario definire le operazioni dello scheduler come regioni critiche.



Un processo P puo' essere in uno dei seguenti stati:

1. In esecuzione. In questo caso il goal **curr_process(X)** leghera' la variabile X a P.
2. Pronto. Questo stato e' rappresentato dal fatto Prolog 'waiting(processor, P).'
3. Sospeso in attesa del verificarsi di un evento E. Questo stato e' rappresentato dal fatto 'waiting(E,P)'.

Lo scheduler implementa le operazioni di creazione, sospensione, riattivazione, e terminazione di un processo e rischedulamento utilizzando la primitiva 'transfer'. Ad esempio le operazioni di sospensione e riattivazione sono definite come segue:

```
suspend(E):-  curr_process(P),
              assert(waiting(E,P)),
              retract(waiting(processor,S)),!,
              transfer(S).
suspend(E):-  idle(X), tranfer(X).
```

Per sospendere il processo corrente P su un evento E, si asserisce che esso attende quell'evento e si trasferisce il controllo al primo processo pronto S. Se non esiste nessun processo pronto, il controllo e trasferito ad un processo "idle".

```
reactivate(E):-
              retract(waiting(E,P)),!,
              assert(waiting(processor,P)).
reactivate(E).
```

Un processo in attesa sull'evento E e' riattivato aggiornando il suo stato da sospeso a pronto.

### 4.2 Una mailbox in Prolog

Le mailbox [17] sono considerate un meccanismo molto generale e flessibile, anche se primitivo, per consentire la comunicazione fra processi. In Prolog le mailbox possono essere rappresentate come unita' contenenti, come messaggi, fatti Prolog del tipo:

```
message(Source,Dest,Item).
```

L' utilizzo di Prolog quale linguaggio di sistema rende l'accesso a mailbox particolarmente flessibile. L'ispezione della coda di messaggi nella mailbox e', infatti, guidata dal meccanismo di unificazione.

Le operazioni di inserimento ('out') e rimozione ('in') di messaggi dalla mailbox possono essere implementate come segue utilizzando lo scheduler precedente:

```
in(E):-  (retract(E),!;
          ask(scheduler,suspend(E)),
          in(E).
```

```
out(E):- assert(E),
         ask(scheduler,reactivate(E)).
```

Le operazioni di 'in' ed 'out' sono di basso livello e presentano effetti collaterali. Esse vanno utilizate solo per programmazione di sistema, allo scopo di definire, per l'utente finale, forme di comunicazione e sincronizzazione di piu' alto livello e con una piu' chiara semantica dichiarativa. In particolare, esse sono state utilizzate per definire, in Prolog, le clausole di sincronizzazione di CPU [6] e gli eventi di Delta- Prolog [14].

### 4.3 Time-sharing

Come gia' detto, l'interprete e' stato modificato in modo da verificare la presenza o meno di una richiesta di interruzione. Al momento di servire tale richiesta, dopo aver salvato il contesto del processo in esecuzione, viene forzata la dimostrazione di un goal specifico chiamato 'interrupt'.
E' stato per questo definito un nuovo predicato di sistema:

```
interrupt :- interrupt_handler, !,exit_handler.
interrupt :- exit_handler.
```

Il predicato **interrupt_handler**, che puo' essere visto come la vera routine di risposta all'interruzione, e' definito dall'utente in funzione del comportamento che vuole ottenere dal sistema. Il predicato di sistema exit_handler ripristina il contesto del processo interrotto.

In caso di fallimento della dimostrazione di interrupt_handler (ovvero la sua non definizione) grazie alla seconda clausola per il predicato interrupt, il processo interrotto continua la sua elaborazione normalmente.

Per ottenere, nel caso di interruzione, il trasferimento del controllo ciclicamente ad un altro processo attivo si puo' definire interrupt_handler come segue:

```
interrupt_handler :- ask(schedule,reschedule).
interrupt_handler.
```

Tale definizione prevede, ovviamente, la presenza nel sistema di uno scheduler analogo a quello descritto nel paragrafo 4.1. Il predicato 'reschedule' da' il controllo al primo processo pronto, dopo aver reso 'pronto' il processo in esecuzione.

## CONCLUSIONI

In questo lavoro sono state presentate alcune estensioni all'interprete C-Prolog per introdurre, in esso, modularita' e concorrenza. Piuttosto che adottare un approccio top-down, cioe' fissare precisi costrutti e politiche e poi modificare la macchina virtuale Prolog sulla base di tali specifiche di alto livello, si e' preferito adottare un approccio bottom-up. Per questo motivo si sono individuati un insieme di nuovi predicati predefiniti sulla base dei quali prototipare velocemente nuove astrazioni utilizzando lo stile dichiarativo di Prolog ed il meccanismo di unificazione. Il Prolog esteso qui presentato e' particolarmente adatto per la programmazione di sistema ed, in quest'ottica, e' utilizzato nell'ambito del Progetto Esprit Alpes [10] quale linguaggio base per la costruzione di un ambiente avanzato di programmazione logica per Prolog integrato ed uniforme.



## RIFERIMENTI


[1] K.Nakashima: "Knowledge Representation in Prolog/KR", International Symposium on Logic Programming, Atlantic City, Febbraio 1984.

[2] T.Chikayama: "ESP Reference Manual", ICOT Report, Feb. 1984.

[3] N.Wirth: "Modula-2 Language Reference Manual", ETH, 1980

[4] K.Bowen, R.Kowalski :"Amalgamating language and metalanguage in logic programming", in Logic Programming , Academic Press, 1982.

[5] R.Venken: "A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query-Optimization", in Proc. of ECAI-84, North Holland, 1984.

[6] P.Mello, A.Natali: "Programs as Collections of Communicating Prolog Units" In: European Symposium on Programming, Saarbrucken, Lecture Notes on Computer Science, Marzo 1986.

[7] M.Bruynooghe: "The Memory Management of Prolog Implementation", In: Logic Programming, Academic Press, 1982.

[8] T.Chikayama: "KL0 Reference Manual", ICOT Tech. Report, 1984.

[9] H.Gallaire: "Merging Objects and Logic Programming: Relational Semantics", AAAI '86, Agosto 1986.

[10] ALPES Esprit Project P973. Technical Annex 1985.

[11] C.Hewitt, P.De Jong :"Open Systems", Tech. Rep. MIT-AIM 691 Dicembre 1981.

[12] H. Kauffman, A. Grumbach: "MULTILOG: MULTIpie worlds in LOGic Programming", ECAI-86, 1986.

[13] K.Clark, S.Gregory :" PARLOG: Parallel Programming in Logic", Research Report DOC 84/4 , Imperial College, 1984.

[14] L.M.Pereira, R.Nasr:"Delta-Prolog: A Distributed Logic Programming Language", International Conference On Fifth Generation Computer Systems, Novembre 1984.

[15] E.Y. Shapiro: "A subset of Concurrent PROLOG and its Interpreter" Technical Report ICOT n.3 , Ottobre 1983.

[16] P.Mello et alii: "Communicating Prolog Units: Specifications of the Basic Prolog Extensions and Their Use", ALPES-Esprit Technical Report, Luglio 1987.

[17] G.R.Andrews, F.B.Schneider: "Concepts and Notations for Concurrent Programming", Computing Surveys, Marzo 1983, Vol.15, No.1.

**APPENDICE A.** Definizione del predicato 'live'.

```
live :-      cycle, halt.
cycle :-     prompt(' ?-'),                    % invio del prompt
             repeat,                           % ciclo di lettura del goal
             read(Goal),
             interpret(Goal).                  % interpretazione
interpret(Goal) :-        var(Goal),!,
                          display('statement is a variable'),
                          fail.
interpret(end_of_file):-  halt.
interpret(Goal) :-        dogoal(Goal), !, fail.
dogoal(Goal) :-   save_read_vars,               % salvataggio variabili
                  call(Goal),
                  repply,                       % stampa dei risultati
                  display(' YES '),             % dimostrazione riuscita
                  reset_read_vars.
dogoal(Goal) :-   display(' NO '),              % dimostrazione fallita
                  reset_read_vars.
```

**APPENDICE B.** Alcune prove di performance relative alle unita' separate.

Si consideri il predicato 'reverse(L,R)' definito come segue:

```
reverse([],[]).
reverse([X1|Xn],R) :-  reverse(Xn,Rn), append(Rn,[X1],R).
append([X1|Xn],Y,[X1|Z]) :-    append(Xn,Y,Z).
append([],X,X).
```

Esso inverte la lista L nella lista R con un numero di inferenze (invocazione di un goal e unificazione con la testa di una clausola) legato al numero N di elementi di L attraverso la funzione $1 + N/2 * (3 + N)$.
Quindi con N = 23 il numero di inferenze eseguite da 'reverse' e' pari a 300. Una valutazione in termini di LIPS (Logical Inferences Per Second) puo' avvenire eseguendo 10 volte tale computazione, calcolando il tempo di CPU utilizzato (in secondi) e dividendo 3000 per esso. Costruendo 100 unita' (u1,...,u100) contenenti tutte la stessa definizione di reverse si considera il caso di massima replicazione dei nomi. Si ottengono i seguenti valori di LIPS in funzione della unita' di lavoro.

| | | | |
|---|---|---|---|
| unita' u1 | 3333 LIPS | unita' u75 | 1682 LIPS |
| unita' u25 | 2535 LIPS | unita' u100 | 1440 LIPS |
| unita' u50 | 2022 LIPS | | |

Consultando poi nella unita' u100 ancora il codice di reverse con un diverso nome del funtore (new_reverse), si ottengono ancora 3333 LIPS in quanto ora u100 e' la prima unita' ad aver definito il predicato new_reverse.



# Un Linguaggio Logico OR-Parallelo per una Classe di Applicazioni di IA

**G. Giandonato, G. Sofi**

CSELT
Centro Studi E Laboratori Telecomunicazioni

Via Reiss Romoli 274 - 10148 Torino

**Sommario**

Questo articolo introduce un linguaggio di programmazione logica parallela orientato ad una classe, specifica ma importante, di problemi di IA : la realizzazione di motori inferenziali complessi (ad es. con ragionamento misto forward/backward, strategie di ricerca guidate da informazioni probabilistiche, etc...) quali quelli tipicamente richiesti in sistemi di interpretazione del segnale basati su regole. Le tre caratteristiche essenziali di questa estensione del Prolog sequenziale - la possibilita' di combinare in modo disciplinato backtracking e parallelismo OR; la capacita' di controllare la strategia di ricerca a livello di selezione del nodo; un meccanismo di cooperazione tra processi basato su asserzione di lemmi - sono descritte ed illustrate mediante un esempio esteso derivato da un'applicazione reale (parsificazione di un reticolo di ipotesi di parola in un sistema di comprensione del parlato naturale continuo).

## 1. Introduzione

Il nostro lavoro puo' essere considerato come un tentativo di conciliare l'elevato potere espressivo dei linguaggi di programmazione logica con le stringenti esigenze di efficienza di alcune tra le piu' interessanti applicazioni di IA (dialogo uomo-macchina in linguaggio naturale, visione artificiale, etc...).
Le applicazioni citate, e piu' in generale quelle di interpretazione del segnale con sistemi basati su regole, sono infatti caratterizzate dalla natura nondeterministica dei dati in ingresso, dovuta sia a rumore sul segnale stesso e/o sul sensore e/o sul canale, sia alla imperfezione della conoscenza (euristica o statistica) solitamente utilizzata nell'elaborazione di primo livello. La grande molteplicita' di questi dati (o fatti) incerti in ingresso al motore inferenziale accoppiata con il nondeterminismo intrinseco delle regole genera un enorme spazio di ricerca, che andra' esplorato in modo non esaustivo e quindi guidato in modo *best-first* da informazioni di priorita', derivate ad esempio dai fattori di certezza dei fatti iniziali osservati, originando differenti sottocomputazioni a partire da differenti sottoinsiemi di tali fatti.
Di conseguenza il sistema deve garantire una adeguata capacita' elaborativa per rispettare i tempi brevi di risposta richiesti dall'applicazione finale (si pensi al caso della comprensione del parlato nel contesto di un sistema di servizi di telecomunicazioni basati su dialogo).
Una ovvia soluzione a questo genere di esigenza e' l'utilizzo di strutture elaborative ad elevato grado di parallelismo; in [BGGM87] abbiamo descritto un'architettura parallela per la classe di applicazioni sopra accennate che impiega uno schema di parallelizzazione