Un Linguaggio Funzionale in Prolog

Andrea Vezzosi 4223606

18 settembre 2012

Sommario

La particolare attitudine alla manipolazione di espressioni simboliche rende Prolog uno strumento molto conveniente per la sperimentazione sui linguaggi di programmazione. L'elaborato presenta l'implementazione di un semplice linguaggio funzionale con polimorfismo parametrico, tipi di dato algebrici e lazy evaluation.

1 Introduzione

In questa sezione descriveremo informalmente il linguaggio, nelle seguenti analizzeremo alcuni aspetti salienti dell'implementazione.

$$
\begin{array}{lcl}
\langle \mathit{expr} \rangle & ::= & \backslash \langle \mathit{ident} \rangle .\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{ident} \rangle \\
 & | & \text{case } \langle \mathit{expr} \rangle \text{ of } \langle \mathit{clauses} \rangle \\
 & & \\
\langle \mathit{clauses} \rangle & ::= & \langle \mathit{pattern} \rangle .\ \langle \mathit{expr} \rangle\ ;\ \langle \mathit{clauses} \rangle \mid \langle \mathit{empty} \rangle \\
 & & \\
\langle \mathit{pattern} \rangle & ::= & \langle \mathit{constructor} \rangle\ \langle \mathit{vars} \rangle \\
 & & \\
\langle \mathit{vars} \rangle & ::= & \langle \mathit{ident} \rangle\ \langle \mathit{vars} \rangle \mid \langle \mathit{empty} \rangle
\end{array}
$$

Figura 1: Grammatica di alto livello

Il nostro linguaggio è principalmente ispirato al Lambda Calculus, come dimostrano le prime tre alternative di *expr* anche se per motivi di facilità di input usiamo \ al posto di λ, in concreto useremo poi parentesi per raccogliere le sotto-espressioni in caso di necessità e un'espressione del tipo $a\ b\ c$ verrà interpretata come $(a\ b)\ c$, ovvero l'applicazione di funzioni associa a sinistra.

Per procedere con degli esempi introduciamo subito alcuni tipi algebrici disponibili, i booleani e i naturali, tramite i loro costruttori:

```
?- repl.
> true
bool
true
> false
bool
false
> zero
nat
0
> suc
nat->nat
$function
```

Nel log precedente la query "*repl.*" ha prodotto l'esecuzione dell'ambiente interattivo per il nostro linguaggio, accettando espressioni dopo il prompt >. Dopo ogni espressione ne viene stampato il tipo e poi il valore; $nat \rightarrow nat$ ad esempio è il tipo delle funzioni dai naturali ai naturali, come ci si aspetta per la funzione successore, mentre il suo valore viene omesso come quello di tutte le funzioni perchè normalmente l'utente dovrebbe considerarle opache. Infine possiamo notare che il valore di *zero* è 0, in effetti supportiamo sia la sintassi unaria che quella decimale per ogni naturale, e per brevità preferiamo la seconda nello stampare i valori.

Se prendiamo ad esempio i termini Prolog ogni tipo di dato algebrico è un sottoinsieme tipato di essi, però nel nostro linguaggio non supportiamo l'unificazione ma solo il pattern matching, come tipico dei maggiori linguaggi esclusivamente funzionali. Con il pattern matching possiamo ad esempio definire e usare la funzione not, che nega un booleano.

```
> let not = \b. case b of true. false; false. true; in not
bool->bool
$function
> let not = \b. case b of true. false; false. true; in not true
bool
false
> let not = \b. case b of true. false; false. true; in not false
bool
true
```

Un altro utile tipo algebrico è quello delle liste.

```
> nil
forall([a],list(a))
nil
> cons
forall([a], (a->list(a)->list(a)))
$function
> cons 1 (cons 2 nil)
list(nat)
cons 1 (cons 2 nil)
> cons true nil
list(bool)
cons true nil
> cons 0 (cons true nil)
type error.
```

I costruttori *nil* e *cons* sfruttano un'altra caratteristica del linguaggio, il loro tipo è infatti polimofico: il tipo delle liste $list(a)$ è parametrizzato da a che è il tipo degli elementi, quindi $forall([a], list(a))$ ci dice che *nil* è in grado di produrre una lista di a per ogni tipo a e allo stesso modo *cons* a patto che gli argomenti concordino.

Uno degli esempi più semplici del Lambda Calculus è la funzione identità, essa è anche un esempio di funzione polimorfica che non fa uso di tipi algebrici.

```
> \x. x
forall([a], (a->a))
$function
> (\x. x) 0
nat
0
```

L'ambiente interattivo può anche caricare definizioni da un file, infatti supponendo di avere il seguente codice nel file "esempi":

```
:let take = \n.\xs.
     case n of
       zero. nil;
       suc n. case xs of
       nil. nil;
cons x xs. cons x (take n xs);;;

:let add = \n.\m.
   case n of
     zero. m;
     suc n. suc (add n m);;
```

```
:let zipWith = \f.\xs.\ys.
   case xs of
     nil. nil;
     cons x xs.
        case ys of
           nil. nil;
           cons y ys. cons (f x y) (zipWith f xs ys);;;

:let tail = \xs.
      case xs of
       cons x xs. xs;;

:let fibs = cons 1 (cons 1 (zipWith add fibs (tail fibs)));
```

possiamo procedere con la seguente interazione dimostrando il supporto per la lazy evaluation. In essa *fibs* è una lista infinita di tutti i numeri della sequenza di fibonacci e ne osserviamo i primi 10 elementi senza causare la non-terminazione, ciò è realizzabile grazie alla strategia di valutazione lazy dove riduciamo un'espressione solo quando non possiamo farne a meno.

```
> :load esempi
[take:forall([a], (nat->list(a)->list(a)))]
[add: (nat->nat->nat)]
[zipWith:forall([a,b,c], ((a->b->c)->list(a)->list(b)->list(c)))
    ]
[tail:forall([a], (list(a)->list(a)))]
[fibs:list(nat)]
> take 10 fibs
list(nat)
cons 1 (cons 1 (cons 2 (cons 3 (cons 5 (cons 8 (cons 13 (cons 21
     (cons 34 (cons 55 nil)))))))))
```

Il file "definizioni" allegato all'elaborato contiene queste e altre definizioni d'esempio.

2 Type-Checker

$$\textsc{Var}\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Lam}\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \beta}$$

$$\textsc{App}\frac{\Gamma \vdash e_0 : \alpha \to \beta \qquad \Gamma \vdash e_1 : \alpha}{\Gamma \vdash e_0\ e_1 : \beta}$$

Figura 2: Simply Typed Lambda Calculus

Per spiegare come il nostro typechecker gestisce il polimorfismo è utile partire dal più semplice Simply Typed Lambda Calculus (Figura 2) che è immediatamente traducibile nel predicato *stlc*. Dobbiamo qui richiedere l'esecuzione dell'occurs check per garantire che non vengano introdotte strutture cicliche.

```
:- set_prolog_flag(occurs_check, true).
:- op(500, yfx, $).

%% stlc(Contesto, Espressione, Tipo)
stlc(C, var(X), T) :- member(X:T, C).
stlc(C, lam(X,E), A -> B) :- stlc([X:A|C], E, B).
stlc(C, E0 $ E1, B) :- stlc(C, E0, A -> B), stlc(C, E1, A).
```

Analizzando le clausole possiamo notare che sono mutuamente esclusive e strutturalmente ricorsive per quanto riguarda il secondo argomento e se assumiamo che le variabili nel contesto siano distinte anche la chiamata di member sarà deterministica: in questo caso quando l'espressione è ground (e finita) Prolog ci darà un'unica soluzione o un fallimento finito.

Un semplice esempio è quello della funzione identità:

```
?- stlc([],lam(x,var(x)),T).
T = (A->A) ;
false.
```

La soluzione $T = (A \to A)$ e la sua unicità ci dicono che all'espressione $lam(x, var(x))$ possiamo assegnare tutti e soli quei tipi che unificano con $A \to A$, *stlc* però non ci dà la possibilità di definire la funzione identità una volta per tutte e riusarla per ogni suo tipo valido:

```
?- stlc([], lam(x,var(x)) $ lam(x,var(x)), T).
T = (_G272->_G272) ;
false.

?- stlc([], lam(id, var(id) $ var(id)) $ lam(x,var(x)), T).
false.
```

La seconda query fallisce perchè l'applicazione di *id* ad *id* sarebbe possible solo se $A = (A \to A)$ che è impedito dall'occurs check, ma il successo della prima query ci dice che se per ogni uso di *id* generassimo nuove variabili da sostituire ad A allora avremmo un successo.

Possiamo sfruttare questa intuizione per definire un nuovo predicato *type* che implementerà quello che in letteratura è conosciuto come polimorfismo alla Hindley-Milner o let-polymorphism:

```
type(C, var(X), T) :- member(X:T0,C),instantiate(T0,T).
type(C, lam(X,E), A -> B) :- type([X:mono(A)|C], E, B).
type(C, E0 $ E1, B) :- type(C, E0, A -> B), type(C, E1, A).
type(C, let(X = E0,E1), T) :-
    type([X:mono(A0)|C], E0, A0), generalize(C, A0, A),
    type([X:A|C], E1, T).

generalize(C, T, poly(Vs,T)) :- term_variables(C, Vs).

instantiate(poly(Vs,T0), T) :- copy_term((Vs,T0), (Vs,T)).
instantiate(mono(T), T).
```

I tipi nel contesto sono adesso classificati come $mono(T)$ (monomorfici) se devono comportarsi come i tipi di *stlc* oppure $poly(Vs, T)$ nel caso in cui le variabili logiche presenti vadano "rinfrescate" ad ogni utilizzo della variabile associata. Il predicato *instantiate* si occupa di questo.

Le variabili introdotte da una λ saranno monomorfiche perchè in generale la funzione così creata non verrà applicata a un singolo argomento di cui possiamo dedurre il tipo più generale.

Introduciamo allora un nuovo construtto sintattico "$let\ x = e_0\ in\ e_1$" il cui valore è quello di e_1 in un ambiente dove x è definita come e_0, in questo modo una volta trovato il tipo più generale di e_0 possiamo associare ad x il corrispondente tipo polimorfico calcolato da *generalize*.

Dall'uso di *generalize* si evince che Vs in $poly(Vs, T)$ raccoglie le variabili logiche presenti nel contesto, esse infatti potranno essere influenzate da parti dell'espressione esterne a e_0 e quindi dovranno mantenere lo stesso valore per ogni uso di x. Di questo si occupa ancora *instantiate*. Infine consentiamo a e_0 di riferisi ricorsivamente a x ottenendo così un linguaggio Turing-completo.

Per quanto riguarda i tipi di dato algebrici possiamo definire il predicato $axiom(Name, Type)$ che associa al nome di ogni construttore il suo tipo. Dato *axiom* è semplice estendere *type* per supportarli e rimandiamo al codice completo per i dettagli.

3 Interprete

Al più alto livello di astrazione la semantica del nostro linguaggio potrebbe essere specificata tramite regole di riscrittura, ma dato che siamo interessati a un particolare ordine di valutazione (lazy) possiamo invece partire da un interprete per un ordine più semplice, call-by-name.

```
first((X:Y), Xs) :- memberchk((X:Y0), Xs), Y0 = Y.

eval(var(X) / Env, V) :- first(X:C, Env), eval(C,V).
eval(lam(X,B) / Env, lam(X,B) / Env).
eval((E0 $ E1) / Env, V) :- eval(E0 / Env, lam(X,B) / LEnv)
                          , eval(B / [X:(E1/Env)|LEnv], V).
```

Le clausole presentate sono sufficienti solo per il frammento del nostro linguaggio corrispondente al lambda calculus puro, ma sono comunque esplicative.

L'interprete è basato sul concetto di chiusura, ovvero di una coppia (E/Env) di un espressione E e l'ambiente Env che racchiude le sue variabili libere e le associa alle chiusure che rappresentano il loro valore.

Se il predicato termina con successo il risultato sarà una chiusura della forma (lam(X,B) / Env) o quando aggiungeremo i tipi di dati algebrici potrà avere un construttore invece di una lambda.

La clausola più interessante è l'ultima, dove possiamo notare che l'argomento E1 non viene valutato prima di valutare il corpo B della funzione ricavata da E0, ma è solo aggiunto inalterato all'ambiente di B come espressione della chiusura relativa ad X. Questo produce gli stessi risultati della lazy evaluation, ma è molto inefficiente: ogni volta che nella valutazione di B avremo bisogno del valore di X eseguiremo eval(E1/Env, V) ricalcolando di nuovo il risultato V invece di ricordarlo dopo la prima volta.

Per ovviare a questo possiamo immagazzinare le chiusure in un heap invece che negli ambienti che conterranno invece l'indice corrispondente. Ciò consentirà di aggiornare l'heap con V una volta calcolato.

```
lookup(R,heap(_,H)) :- first(R,H).

update(Ref, Value, heap(N,H0), heap(N,H)) :-
      append(Xs,[Ref:_|Ys],H0)
   -> append(Xs,[Ref:Value|Ys],H)
   ;  H0 = H.

alloca(C, Ref, heap(Ref,H), heap(Next,[Ref:C|H]))
   :- Next is Ref + 1.

eval(var(X) / Env, V, H0, H) :-
```

```
  first(X:Ref, Env), lookup(Ref:C, H0),
  eval(C, V, H0, H1), update(Ref, V, H1, H).
eval(lam(X,B) / Env, lam(X,B) / Env, H, H).
eval((E0 $ E1) / Env, V, H0, H) :-
  eval(E0 / Env, lam(X,B) / LEnv, H0, H1),
  alloca(E1 / Env, Ref, H1, H2),
  eval(B / [X:Ref|LEnv], V, H2, H).
```

eval implementa adesso correttamente la strategia di valutazione lazy, ed è completato con le clausole che gestiscono il resto dei construtti del nostro linguaggio nel codice dell'elaborato.

Per raggiungere un livello di efficienza accettabile si deve però anche fare attenzione a eliminare i punti di scelta superflui: ad esempio invece di clausole con teste della forma eval(E / Env, V) converrà definirle della forma eval(E , Env, V) in modo che l'interprete swi, indicizzando sul primo argomento, faccia uso della loro mutua esclusività.

4 Parser e Pretty-Printer

Supponendo di avere come input una lista di token implementiamo un parser per il nostro linguaggio come una Definite Clause Grammar *expr*. Le keyword sono rappresentate come atomi, mentre i(X) è il token che rappresenta la variabile di nome X.

```
expr(lam(X,E)) --> [\], [i(X)], [.], expr(E).
expr(let(B,E)) --> [let], binding(B), [in], expr(E).
expr(case(E,Cs)) --> [case], expr(E), [of], clauses(Cs).
expr(E) --> spine(E).
```

binding e *clauses* sono DCG per frammenti rispettivamente del tipo "$x = e$" e "$pattern_0$. e_0; ... ; $pattern_n$. e_n;" e la loro implementaione non pone problemi.

spine ha il compito di produrre l'albero sintattico per esperessioni della forma "$t_0 \dots t_n$" dove ogni t_i è una singola variabile o un'espressione fra parentesi.

```
term(var(X)) --> [i(X)].
term(E) --> ['('],expr(E),[')'].

spine(E $ T) --> spine(E), term(T).
spine(E) --> term(E).
```

Questa definizione però non è adeguata perchè la ricorsione a sinistra nella prima clausola di *spine* causerebbe un loop infinito. Potremmo utilizzare le note tecniche per eliminare la ricorsione a sinistra da una grammatica, ma questo produrrebbe una DCG che non terminerebbe quando volessimo usarla nella direzione inversa, ovvero come pretty-printer.

Una soluzione è definire una DCG bidirezionale che fa il parse di "$t_0 \ldots t_n$" come una lista non-vuota e poi trasformarla nell'albero sintattico corrispondente: essenzialmente una lista associata a sinistra invece che a destra.

```
terms([T|Ts]) --> term(T),terms(Ts).
terms([T]) --> term(T).

fold([Tm|Tms], F, Tree, G $ _) :- fold(Tms, F $ Tm, Tree, G).
fold([], Tree, Tree, _).

fold([X|Xs], E) :- fold(Xs, X, E, E).
```

Il predicato $fold/2$ si occupa del cambio di associamento ed è implementato con la tecnica dell'accumulatore tramite $fold/4$. Quest'ultimo è strutturalmente ricorsivo nel primo argomento, quindi termina quando la lista di termini è ground, ma è anche strutturalmente ricorsivo nel quarto argomento e quindi termina anche quando è l'albero sintattico a essere ground, grazie a questo accorgimento otteniamo la bidirezionalità.

spine viene quindi definito come la composizione dei precedenti predicati, l'ordine di esecuzione deve però tenere conto di quale fra l'albero sintattico e la lista di token è l'input ground.

```
spine(E) --> {var(E)} -> terms(Ts), {fold(Ts,E)}
                      ;  {fold(Ts,E)}, terms(Ts).
```

In questo modo abbiamo ottenuto un solo predicato *expr* che possiamo utilizzare sia come pretty-printer che come parser, evitando così ripetizioni e conseguenti possibilità di errore nel nostro codice.

Nella pratica i predicati di I/O utilizzano liste di caratteri, ma possiamo facilmente definire due predicati che li convertano da e in liste di token.