Aplicación de la evaluación parcial a la metaprogramación

Juan González

Report LSI–90–20

Aplicación de la Evaluación Parcial a la Metaprogramación

Juan González
Departamento de Software (Lenguajes y Sistemas Informáticos)
Universidad Politécnica de Cataluña
08028 Barcelona, SPAIN

Abstract:

If we apply partial evaluation to a meinterpreter and a program, we obtain another program that combines the functionalities of interpreter and the results of the program, without the inefficiency of the interpretation.
We are studying three basic Prolog metainterpreters: memoization, lazy evaluation and transformation.
The results from these examples are very satisfactory and confirm the usefulness of a general environment to the disposal of the programmer in order to find through an experimental way, efficient versions of their program.

Resumen:

Si aplicamos un evaluador parcial a un metainterprete y un programa, obtenemos otro programa que combina la funcionalidad del interprete y los resultados del programa, sin la ineficiencia de la interpretación.
Si la funcionalidad del interprete tiende a mejorar el control del programa origen, el programa resultante puede ser mucho más eficiente.
Estudiamos tres metainterpretes de Prolog básicos : memorización , evaluación lazy y transformación . Los resultados sobre ejemplos son muy satisfactorios y confirman la utilidad de un entorno general a disposición del programador para que de forma experimental encuentre versiones eficientes de su programa.

1. Introducción

Evaluación Parcial

En programación lógica la evaluación parcial es la evaluación en tiempo de compilación de un objetivo con algunos de sus parámetros instanciados o parcialmente instanciados.
Un evaluador parcial *peval* toma un objetivo Q relativo a un programa P y produce un nuevo programa P´ , que no es más que la especialización de P sobre Q. Para el objetivo Q , P y P´ computan las mismas soluciones , pero P´ es más eficiente.
Los evaluadores parciales se utilizan como herramientas de :optimización de programas , generación de compiladores y metaprogramación.

Metaprogramación

La metaprogramación es una metodología de construcción de programas que en vez de proponer soluciones particulares , construye un lenguaje orientado al problema , de manera que el usuario puede actuar interactivamente expresando una gran variedad de requerimientos computacionales , y también le permite experimentar y comprobar efectos complicados de preveer durante la especificación .

Esta visión proporciona las muy conocidas ventajas de :modularidad , claridad, modificabilidad. Pero tiene un inconveniente fundamental , su ineficiencia.
Es aquí precisamente donde tiene interés utilizar la evaluación parcial para evitar el coste de la interpretación del nuevo lenguaje.

Proponemos tres ampliaciones del lenguaje de programación Prolog, ya sugeridas en [3], en orden a favorecer la construcción de un entorno de experimentación de diferentes controles para la ejecución de programas.
De forma general si L es un lenguaje y construimos una extensión del lenguaje L+ de L, con autointerprete Int+, tendremos que todo programa P de L+ puede trasladarse a L utilizando peval(Int+ (P)), sin sobrecargar su eficiencia.

Las ampliaciones las designaremos de forma abreviada :

Memorización
Lazy
Transformación

2. Modificación del Lenguaje de Programación

Es importante determinar el tipo de control óptimo de un programa. Para eso disponemos de un catálogo de simuladores de distintos controles. Estos simuladores son interpretes capaces de modificar la evaluación normal de objetivos añadiendo alguna habilidad que aproveche ciertas propiedades del programa, para acelerarlo.

Una primera aproximación nos lleva a metainterpretes simples : de memorización (que elimine computaciónes redundantes), de evaluación lazy (que resuelva problemas de terminación y evite la utilización de estructuras de datos innecesarias) y de transformación (que permita utilizar teoremas de equivalencia de programas).
Una continuación razonable sería facilitar la composición de controles más complejos mediante combinación de estos más sencillos.

2.1. Interprete de Memorización

En este caso la ampliación está orientada a evitar la evaluación de objetivos ya evaluados anteriormente.Una solución inicial sería memorizar todos los objetivos ya evaluados. La mejora de esta solución pasaría por resolver dos cuestiones básicas :

Memorización de los objetivos que son estrictamente necesarios en cada momento.
Consulta eficiente de los objetivos memorizados.

Una lista puede servir de forma sencilla y suficiente para mostrar el efecto simplificador de la evaluación parcial. De manera que el interprete para resolver un objetivo , mira si éste está en la lista de memorización , si es así toma su valor, y si no procede a su evaluación normal.

```
solve (G, M, NM )
        El objetivo es resoluble con la lista de objetivos M ya memorizados , obteniendo la
nueva lista de objetivos NM.

solve ((G,RG), M , NM) :-
        solve (G, M, TM),
        solve (RG, TM, NM).
solve (G, M, M) :-
        sistema (G),
        G.
```

```
solve (G,M, M) :-
        not (sistema (G)),
        member (G,M).
solve (G, M, NM) :-
        not (member (G,M))
        clause (G,T),
        solve(T, [G/M],NM).
```

Otra solución alternativa utiliza la base de datos y los predicados predefinidos assert y retract. Sin embargo esto complica la utilización de evaluación parcial, por el carácter extra lógico de assert y retract.

```
solve ((G,RG)) :-
        solve (G),
        solve (RG).
solve (G) :-
        sistema (G),
        G.
solve (G) :-
        not (sistema (G)),
        m(G).

solve (G) :-
        not (m(G)),
        clause (G,RG),
        solve (RG),
        miasserta (m(G)).

miasserta (G) :-
        asserta (G).
miasserta (G) :-
        retract (G).

sistema (true).
sistema ( _ is _).
sistema ( _ < _).
sistema ( _ > _).
sistema ( not (X)).
sistema (clause (X,Y)).
```

2.2.Interprete Lazy

Se quiere ampliar el lenguaje con el predicado freeze(Variable, Objetivo) de manera que el objetivo esté inactivo hasta que la variable sea instanciada.
Los programas con estructuras del tipo comprobador-generador son especialmente adecuados para aplicarles este tipo de evaluación. La idea es desactivar el comprobador hasta que el generador produzca algún avance en la solución.

Una extensión de la solución dada por [1] es :

solve (G, F, NF)
Se puede resolver el objetivo G a partir de la lista de objetivos inactivos F, obteniendo una nueva lista de objetivos NF.

```
solve ((G, RG), F, NF) :-
        solve (G, F, TF),
        solve (RG, TF, NF).

solve(G,F,F) :-
        sistema (G),
        G.

solve (G, F, NF) :-
        not (sistema (G)),
        clause (G, T),
        defrost (T,TF),
        solve (T,TF, NF).


solve (freeze(X,G), F , [[X/G]/F]) :-
        var(X).

solve (freeze(X,G), F, NF ) :-
        nonvar(X),
        solve(G,F,NF).

defrost ([], []).
defrost([[X/G]/F],[[X/G]/NF]):-
        var(X),
        defrost(F,NF).

defrost([[X/G]/F], NF) :-
        nonvar(X),
        defrost(F,TF),
        solve(G,TF,NF).


sistema (true).
sistema ( _ is _).
sistema ( _ < _).
sistema ( _ > _).
sistema ( not (X)).
sistema (clause (X,Y)).
```

2.3. Interprete de Transformación

Este interprete es un sistema de transformación, que utiliza un catálogo de transformaciones de programas, caracterizadas por preservar el significado del programa inicial , pero mejorando su eficiencia, es una adaptación de las ideas de [2] a Prolog.
Justificamos su utilización como interprete, por la necesidad de representar los esquemas de programa a través de listas, de manera que la ejecución de los programas obtenidos pasa por una interpretación.

La interpretación de un programa consistirá en :
reconocimiento de su estructura
transformación correspondiente si las precondiciones lo permiten
interpretación del programa transformado

Una versión simplificada del interprete sería:

```
solve (Objetivo, Pfuente) :-
        transformacion (Pfuente, Pdestino, Precond),
        solvaux (Objetivo, Pdestino).

tranformacion (Pfuente, Pdestino) :-
        trans (Pfuente, Pdestino, Precond),
        verifica (Precond), !.
transformacion (Pfuente, Pfuente).

solvaux (true, P).
solvaux ((O, Os), P) :-
        solvaux (O, P),
        solvaux (Os, P).
solvaux (O,P) :-
        clausula (O, c (H,B), P),
        solvaux (B,P).
```

3. Evaluador Parcial Peval

Nuestro objetivo es la evaluación parcial de los interpretes anteriores , en orden a optimizar su utilización, sin pretender mejorar el interprete o el programa inicial.
Los resultados básicos de la aplicación de peval , de forma parecida a [4],[5],[6] , son :
Poda de las reglas no utilizadas en la resolución de solve (Objetivo)
Propagación hacia adelante de valores a través de la unificación
Evaluación de predicados del sistema
Para que peval sea operativo y no haya problemas de terminación durante la evaluación parcial necesitamos una lista de Objetivos ya visitados .

peval (G, RG, M, NM)
La evolución parcial del objetivo G es el objetivo residual RG, siendo M la lista de subobjetivos evaluados anteriormente y NM la nueva lista de subobjetivos.

```
peval ((G,Gs), (RG, RGs), M, NM) :- !,
        peval (G, RG, M, M1),
        peval (Gs, RGs, M1, NM).

peval (G, R, M, M) :-
        sistema (G), !,
        pevalS (G,R).

peval (G, R, M, NM ) :-
        posible_unfold (G,M), !,
        clause (G, BG),
        peval (BG, R, [G/M], NM).

peval (G, R, M, M) .
```

```
pevalS (G, RG)
        El objetivo del sistema G , produce el objetivo residual RG.

pevalS (true, true).
pevalS (sistema (A), true) :-
        sistema (A), !.
pevalS (A is B, A is B):-
        var (A).
pevalS (A < B, A < B) :-
        var (A).
pevalS (A < B, A < B) :-
        var (B).
pevalS (A > B, A > B) :-
        var (B).
pevalS (A >B, A > B) :-
        var (A).
pevalS (not (member (X,Y)), not (member (X,Y))) :- !.

pevalS ( not (A), not (A)) :-
        var (A).
pevalS ( not (A), true ) :-
        nonvar (A),
        not (A), !.
pevalS (clause (A,B), true ) :-
        clause (A,B).


posible_unfold (G, M)
        G es desarrollable en el contexto M.

posible_unfold (G, M) :-
        not (member (G, M)),
        functor (G, P, N),
        arg (1, G, A),
        predicado (P),
        argumento (A).
predicado (peval).
predicado ( solve).
predicado (sistema).
argumento (true).
argumento ((A,B)).
argumento (true).
argumento (A) :-
        sistema (A).
```

4. Ejemplos

Estudiamos dos ejemplos muy significativos de : memorización y evaluación lazy, mostrando los programas transformados.

Ejemplo de memorización

En este ejemplo se puede ver que fib(N2, F2) ya ha sido calculado al obtener fib(N1,F1), luego la memorización de objetivos será muy útil.

```
fib(0,1).
fib(1,1).

fib(N,F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib (N1, F1),
        fib (N2, F2),
        F is  F1  +  F2.
```

El programa obtenido aplicando peval al objetivo solve (fib(X,Y), Z,T) es:

```
solve (fib(X,Y), Z,Z) :-
        member (fib(X,Y), Z).
solve (fib(0,1), Z, [fib(0,1)/ Z]) :-
        not( member (fib (0,1), Z).
solve (fib(1,1), Z, [fib(1,1)/ Z]) :-
        not( member (fib (1,1), Z).
solve (fib(X,Y), Z, [fib (X,Y) /NZ]) :-
        not (member (fib (X,Y), Z)),
        X > 1,
        X1 is X - 1,
        X2 is X -2,
        solve (fib (X1,Y1), Z, Z1),
        solve (fib (X2, Y2), Z1, NZ).
```

Este programa residual hereda la conducta de solve sobre fib, pero sin la penalización de tener que analizar estructuralmente el objetivo actual.

Ejemplo evaluación lazy

El ejemplo no es muy realista , pero si muy claro para ilustrar el conflicto entre una especificación fácil y un costo innecesario al utilizar la lista L.
Se trata de escribir el intervalo de enteros [N,M].

```
escribir_generar :-
        freeze(L, escribir_lista(L)),
        generar_lista (1, 10, L).

generar_lista (N, N, [N]).
generar_lista (N, M, [N/G1]) :-
        generar_lista (N1, M, G1).


escribir_lista([]) .
escribir_lista ([H/T]) :-
        freeze(H, write(H)),
        freeze (T, escribir_lista (T)).
```

El programa residual sería :

```
solve( genera_escribe (X,X), F, F) :-
        write (X).

solve (genera_escribe (X,Y), F, F ) :-
        write (X),
        X1 is X + 1,
        solve (genera_escribe (X1, Y)).
```

5. Estimación de la simplificación producida por el Evaluador Parcial

Suponemos que la estructura básica que permite calcular el coste de un objetivo es su árbol de búsqueda.
Sea G un objetivo, P el programa ejemplo , Int un interprete abstracción o generalización de los vistos anteriormente, R el programa residual y Peval el evaluador parcial.
Si queremos establecer la capacidad simplificadora de Peval, tenemos que relacionar los árboles de búsqueda de Int(G,P) y R(G).
El árbol de búsqueda de Int(G,P) se obtiene del árbol de G respecto a P, de manera que cada nodo no terminal es sustituido por un subárbol de profundidad aproximada $p = K.n$, siendo K una constante y n una estimación del número medio de subobjetivos en las clausulas de P.
Si observamos la conducta de peval vemos que recorre el arbol de búsqueda de Int(G,P) y elimina en el peor de los casos los subarboles añadidos en la fase de interpretación , ya que los considera objetivos del sistema y por tanto simplificables.
Resumiendo el árbol de R(G) es más simple o igual que el de G sobre P.

6. Análisis comparativo de los tiempos de ejecución para cada ejemplo

Sobre un VAX 8600 , utilizando un interprete de PROLOG de Edinburgo, y para los programas ejemplo, interpretes y programas residuales anteriores organizados en

nivel objetivo :	fib(_,_),
nivel interpretación + objetivo:	solve (fib(_,_), _, _), solve (genera_escribe(_,_), _, _)
nivel residual :	solve1 (fib(_,_), _, _), solve1 (genera_escribe(_,_), _, _)

Se han obtenido los siguientes tiempos de ejecución:

X	fib(X,Y)	solve(fib(X,Y), [],Z)	solve1(fib(X,Y),[],Z)
3	0.01	0.05	0.02
5	0.01	0.15	0.03
10	0.02	2.98	0.04
15	0.39	102.44	0.06

Para X > 15 los niveles de recursión se hacen inaceptables y fib(15,Y) no acaba, sin embargo solve y solve1 siguen funcionando.
Se confirma la notable mejora conseguida con solve1(fib(_,_),_,_).

X	solve (genera_escribe (1,X),_,_)	solve1(genera_escribe(1,X),_,_)
5	0.23	0.01
10	0.43	0.01
20	1.16	0.01
30	2.29	0.03

7. Conclusiones y ampliaciones futuras

Proponemos la construcción de un entorno que permita al programador , mediante experimentación encontrar el tipo de control adecuado a la especificación de su problema.
Un primer paso inicial sería disponer de metainterpretes básicos : memorización, lazy, transformación, y luego por composición controles más sofisticados.
La evaluación parcial permite que el programa residual herede el control del interprete y el comportamiento del programa inicial , sin acusar los costes de la interpretación.
Los resultados obtenidos en los ejemplos confirman ampliamente estas suposiciones.
Sin embargo hay muchas cuestiones parcialmente resueltas debido a la simplificación del planteamiento.

Sería interesante que :

Hubiera más interpretes básicos de interés.
El evaluador parcial fuera más completo y autoaplicable
Se definan los mecanismos de combinación de interpretes simples para obtener otros más complejos.

Referencias

[1] Cohen J., Describing Prolog by its Interpretation and Compilation, Communications of the ACM, Vol 28, Nº 12, 1985.

[2] Darlington J., Burstall R. M., A System which Automatically Improves Programs, Programming Methodology, Springer-Verlag p. 176-197.

[3] Jones N., Towards Automating the Tranformation of Language Specifications into Compilers, DIKU, University of Copenhagen, 1987.

[4] Levi, G., Sardu G., Partial Evaluation of Metaprograms in a Multiple Worlds Logic Language New Generation Computing, Vol. 6 , 1988.

[5] Sterling L., A meta-level architecture for expert systems, Meta-level Architectures and Reflection , (North-Holland) 1988.

[6] Sterling L., Beer R.D., Metainterpreters for expert system construction, J.Logic Programmming, 1989.

[7] Sterling L., Beer R. D., Incremental Flavor_Mixing of Meta-Interpreters for Expert System Construction, SLP , Salt Lake City, Utah 1986, p. 20-27.