Segunda práctica - CPU Anular

Programación Declarativa: Lógica y Restricciones

Jaime Bautista Salinero; 150103

Table of Contents

codigo

Este módulo representa una CPU anular donde cada registro puede contener un único símbolo.

La CPU tiene la capacidad de ejecutar dos tipos de instrucciones:

1. move(i): Copia el contenido del registro `ri` en el registro `ri+1` para 1 <= i < n, y de `rn` a `r1` para i=n.
2. swap(i,j): Intercambia el contenido de los registros `ri` y `rj` para i < j.

Aclaraciones sobre predicados

ejecutar_instruccion

Para este predicado se ha emplado un predicado auxiliar, elaborando un camino diferente según el tipo de la instrucción, move o swap. Así mismo, se han elaborado los caminos correspondientes para que, en el caso en el que se sepan los estados inicial y finales, el predicado sea capaz de deducir la instrucción correspondiente que lo llevará al estado final.

Cada tipo de instrucción solo tendrá su propio camino independiente al otro tipo.

generador_de_codigo

Una de las maneras existentes para que este predicado pueda devolver el camino más corto que lo lleve a la solución es relizar la búsqueda en anchura en vez de en profundidad, siedo está último la opción por defecto para el sistema Ciao prolog.

A pesar de tener la posibilidad de emplear la librería bf de Ciao, se ha implementado una serie de predicados que realizan la búsqueda en anchura. La memoria necesaria para almacenar los estados visitados, así como el camino que los ha llevado hasta ahí, se ha implementado mediante una estructura dinámica, insertando los estados correspondientes a medida que se visiten y eliminado los caminos anteriores. La estructura es la siguiente:

`[[EstadoActual,[Ruta,EstadoInicial]]`.

Ejemplo: `[[regs(1,1),[move(1),regs(1,2)]],[regs(2,2),[move(2),regs(1,2)]],...]`.

Estructura de la documentación

Los predicados explicados en la sección 'Documentation on exports' están ordenados de la siguiente manera para facilitar su estructura:

1. Predicados pedidos en el ejercicio y a continuación sus predicados auxiliares.
2. Predicados sobre listas.

Consultas

Las consultas de comprobación se han automatizado en el código con predicados test.

Generación de la documentación

Esta documentación ha sido generada automaticamente con la herramienta **lpdoc** (`http://ciao-lang.org/ciao/build/doc/lpdoc.html/`). Para generarla, desde una línea de comandos ubicada en el directorio donde se encuentra el fichero de código, se ha ejecutado:

```
~$ lpdoc -t pdf codigo.pl
```

Usage and interface

- **Library usage:**
  `:- use_module(/home/jaime/cpu_anular/codigo.pl).`
- **Exports:**
  - *Predicates:*
    `alumno_prode/4`, `eliminar_comodines/3`, `ejecutar_instruccion/3`, `generador_de_codigo/3`, `remove_wildcard/3`, `substitute_wildcard/2`, `create_symbol_list/4`, `execute_instruction/3`, `execute_instruction1/5`, `execute_instruction1/6`, `subcopy_term/3`, `code_generator/3`, `seen/1`, `add_to_seen/1`, `remove_seen/2`, `next_states/2`, `next_states_list/2`, `bfs/3`.
  - *Properties:*
    `list/1`.

Documentation on exports

alumno_prode/4: PREDICATE

No further documentation available for this predicate.

eliminar_comodines/3: PREDICATE

Usage: `eliminar_comodines(Registros,RegistrosSinComodines,ListaSimbolos)`

Sustituye los comodines * por variables. Será cierto si `RegistrosSinComodines` es una estructura de tipo reg/n que resulta de sustituir los comodines que aparecen en el argumento `Registros`/n por variables. `ListaSimbolos` es una lista que contiene todos los símbolos utilizados en el término `Registros`/n en el mismo orden en los que estos aparecen en los registros, permitiéndose que haya símbolos repetidos.

```
eliminar_comodines(Registros,RegistrosSinComodines,ListaSimbolos) :-
    functor(Registros,regs,N),
    functor(RegistrosSinComodines,regs,N),
    remove_wildcard(N,Registros,RegistrosSinComodines),
    create_symbol_list(0,N,Registros,ListaSimbolos).
```

ejecutar_instruccion/3: PREDICATE

Usage: `ejecutar_instruccion(EstadoActual,Instruccion,EstadoSiguiente)`

Materializa la transición entre los estados actual y siguiente mediante la ejecución de una instrucción.

```
ejecutar_instruccion(EstadoActual,Instruccion,EstadoSiguiente) :-
    execute_instruction(EstadoActual,Instruccion,EstadoSiguiente).
```

generador_de_codigo/3: PREDICATE

Usage: `generador_de_codigo(EstadoInicial,EstadoFinal,ListaDeInstrucciones)`

Será cierto si `ListaDeInstrucciones` unifica con una lista de instrucciones de la CPU que aplicadas secuencialmente desde un estado inicial de los registros representado por

`EstadoInicial` permite transitar hacia el estado final de los registros representado por `EstadoFinal`.

```
generador_de_codigo(EstadoInicial,EstadoFinal,ListaDeInstrucciones) :-
    eliminar_comodines(EstadoInicial,InicialSinComodines,_1),
    eliminar_comodines(EstadoFinal,FinalSinComodines,_2),
    code_generator(InicialSinComodines,FinalSinComodines,ListaDeInstrucciones).
```

remove_wildcard/3: PREDICATE

Usage: `remove_wildcard(I,R,Rs)`

`Rs` será una estructura de datos del mismo tamaño que `R` sustituyendo los comodines(*) de `R` por la variable anónima. `I` será el acumulador, siendo necesario pasarle como primer valor el tamaño total de la estructura de datos.

```
remove_wildcard(0,_1,_2) :- !.
remove_wildcard(I,R,Rs) :-
    arg(I,R,X1),
    substitute_wildcard(X1,X2),
    arg(I,Rs,X2),
    I1 is I-1,
    remove_wildcard(I1,R,Rs).
```

substitute_wildcard/2: PREDICATE

Usage: `substitute_wildcard(X,Y)`

Para el caso en que `X` sea un comodí, `Y` será la variable anónima. En otro caso `Y` será igual que `X`.

```
substitute_wildcard(*,_1).
substitute_wildcard(X,X) :-
    X\== *.
```

create_symbol_list/4: PREDICATE

Usage: `create_symbol_list(I,Max,R,S)`

Este predicado generará una lista, en `S` compuesta por los símbolos de la estructura de datos en `R`omitiendo el comodí (*). En `Max` se le debe de pasar la longitud de la estructura `R` e `I` será la variable usada como acumulador.

```
create_symbol_list(I,Max,_1,S) :-
    I=:=Max,
    list(S),
    !.
create_symbol_list(I,Max,R,[S|Ss]) :-
    I1 is I+1,
    arg(I1,R,S),
    S\== *,
    create_symbol_list(I1,Max,R,Ss).
create_symbol_list(I,Max,R,S) :-
    I1 is I+1,
    arg(I1,R,X),
    X== *,
    create_symbol_list(I1,Max,R,S).
```

execute_instruction/3: PREDICATE

Usage: `execute_instruction(Ea,I,Es)`

Predidado auxiliar para ejecutar instrucción, mediante el cual, al ejecutar la instrucción `I` sobre la estructura de registros `Ea` se debe obtener `Ea`.

```
execute_instruction(Ea,move(I),Es) :-
    nonvar(I),
    functor(Ea,regs,N),
    I=:=N,
    functor(Es,regs,N),
    arg(I,Ea,X1),
    arg(1,Es,X1),
    subcopy_term(N,Ea,Es).
execute_instruction(Ea,move(I),Es) :-
    nonvar(I),
    I1 is I+1,
    I>=1,
    functor(Ea,regs,N),
    functor(Es,regs,N),
    I<N,
    arg(I,Ea,X1),
    arg(I1,Es,X1),
    subcopy_term(N,Ea,Es).
execute_instruction(Ea,swap(I,J),Es) :-
    nonvar(I),
    I>=1,
    functor(Ea,regs,N),
    functor(Es,regs,N),
    I=<N,
    I<J,
    arg(I,Ea,Xi),
    arg(J,Ea,Xj),
    arg(I,Es,Xj),
    arg(J,Es,Xi),
    subcopy_term(N,Ea,Es).
execute_instruction(Ea,move(I),Es) :-
    var(I),
    functor(Ea,regs,N),
    execute_instruction1(Ea,move(0),I,N,Es).
execute_instruction(Ea,swap(I,J),Es) :-
    var(I),
    var(J),
    functor(Ea,regs,N),
    execute_instruction1(Ea,swap(0,1),I,J,N,Es).
```

execute_instruction1/5: PREDICATE

Usage: `execute_instruction1(Ea,In,I,N,Es)`

Predicado encargado de realizar el backtracking correspondiente para averiguar la instrucción `In` de move, que llevar el registro `Ea` a `Es`. `N` será la longitud máxima de los registros mientras que `I` será el acumulador que permite la construcció de todos los movimientos posibles.

```
execute_instruction1(Ea,move(I),I,_1,Es) :-
    execute_instruction(Ea,move(I),Es).
execute_instruction1(Ea,move(I1),I,N,Es) :-
    nonvar(I1),
    I1=<N,
    I2 is I1+1,
    execute_instruction1(Ea,move(I2),I,N,Es).
```

execute_instruction1/6: PREDICATE

Usage: `execute_instruction1(Ea,In,I,J,N,Es)`

Predicado encargado de realizar el backtracking correspondiente para averiguar la instrucción `In` de swap, que llevar el registro `Ea` a `Es`. `N` será la longitud máxima de los registros mientras que `I` y `J` serán los acumuladores que permiten la construcció de todos los cambios posibles. En un camino se incrementará `J` y en otro camino `I` y `J` a la vez.

```
execute_instruction1(Ea,swap(I,J),I,J,_1,Es) :-
    I\==J,
    execute_instruction(Ea,swap(I,J),Es).
execute_instruction1(Ea,swap(I1,J1),I,J,N,Es) :-
    nonvar(I1),
    nonvar(J1),
    I1<J1,
    J1=<N,
    J2 is J1+1,
    execute_instruction1(Ea,swap(I1,J2),I,J,N,Es).
execute_instruction1(Ea,swap(I1,J1),I,J,N,Es) :-
    nonvar(I1),
    nonvar(J1),
    I1<J1,
    J1=<N,
    I2 is I1+1,
    J2 is J1+1,
    execute_instruction1(Ea,swap(I2,J2),I,J,N,Es).
```

subcopy_term/3: PREDICATE

Usage: `subcopy_term(I,Ea,Es)`

Copia los elementos de la estructura `Ea` a la misma posición en la estructura `Es` siempre y cuando el elemento correspondiente en `Es` no esté previamente inicializado. `I` será el acumuludar para recorrer las estructuras y que como valor inicial se le debe de pasar la longitud de las mismas.

```
subcopy_term(0,_1,_2).
subcopy_term(I,Ea,Es) :-
    arg(I,Es,X1),
    var(X1),
    arg(I,Ea,X1),
    I1 is I-1,
    subcopy_term(I1,Ea,Es).
subcopy_term(I,Ea,Es) :-
    arg(I,Es,X1),
```

```
    nonvar(X1),
    I1 is I-1,
    subcopy_term(I1,Ea,Es).
```

code_generator/3: PREDICATE

Usage: `code_generator(Ei,Ef,Path)`

Inicializa la estructura dinámica empleada para la memoria del camino y los estados de la búqueda por anchura. Así mismo, ejecuta la búsqueda por anchura pasandole como punto de partida la estructura inicial `Ei` y como meta la estructura final `Ef`.

```
code_generator(Ei,Ef,Path) :-
    retractall(seen(_1)),
    bfs(Ef,[[Ei,[Ei]]],[_2|Path]).
```

seen/1: PREDICATE

No further documentation available for this predicate. The predicate is of type *dynamic*.

add_to_seen/1: PREDICATE

Usage: `add_to_seen(E)`

Dada una lista de estados `N` los añade a la estructura dinámica de estados ya vistos.

```
add_to_seen([]).
add_to_seen([[N|_1]|Rest]) :-
    assertz(seen(N)),
    add_to_seen(Rest).
```

remove_seen/2: PREDICATE

Usage: `remove_seen(E,L)`

Dada una lista `E` elimina los estados que ya han sido vistos de la estructura dinámica y devuelve una lista nueva con los que no se han visto aún en `L`.

```
remove_seen([],[]).
remove_seen([[N|_1]|Rest],Result) :-
    seen(N),
    !,
    remove_seen(Rest,Result).
remove_seen([State|Rest],[State|Result]) :-
    !,
    remove_seen(Rest,Result).
```

next_states/2: PREDICATE

Usage: `next_states(X,Y)`

Mediante un estado actual y su ruta asociada en `X` y con estructura [Estado,Ruta], se devuelve un estado siguiente con la nueva ruta asociada en `Y` con estructura [NuevoEstado,[NuevoMoviento,Ruta]].

```
next_states([N,Path],[NewN,[Function|Path]]) :-
    ejecutar_instruccion(N,Function,NewN).
```

next_states_list/2: PREDICATE

Usage: `next_states_list(State,Result)`

Devuelve todos los elementos del siguiente nivel en el árbol partiendo del estado actual `State`. `Result` tendrá la misma estructura que los elementos almacenados en la estructura dinámica seen/1.

```
next_states_list(State,Result) :-
    findall(X,next_states(State,X),Result).
```

bfs/3: PREDICATE

Usage: `bfs(Goal,Queue,Path)`

Ejecuta el algoritmo de búsqueda en anchura para una meta `Goal` y una cola en `Queue`. Devuelve el resultado final de la meta y la ruta en `Path`.

```
bfs(Goal,[[Goal,Path]|_1],FinalPath) :-
    !,
    reverse(Path,FinalPath).
bfs(Goal,[State|Rest],Result) :-
    next_states_list(State,Successors),
    remove_seen(Successors,NewStates),
    add_to_seen(NewStates),
    append(Rest,NewStates,Queue),
    bfs(Goal,Queue,Result).
```

list/1: PROPERTY

Usage: `list(X)`

`X` es una lista.

```
list([]).
list([_1|Xs]) :-
    list(Xs).
list([]).
list([_1|L]) :-
    list(L).
```

Documentation on imports

This module has the following direct dependencies:

– *Application modules:*

`classic_predicates`, `datafacts_rt`, `dynamic_rt`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `dynamic`, `datafacts`.

References

(this section is empty)