LDL++

May 3rd, 2002

Michael Minock

LDL

LDL stands for *Logic Data Language*

- A rule based extension to a domain calculus based language
- Described in book
- Developed at MCC, Austin, TX.
- LDL++ developed at UCLA

Implementation Choices

- Not based on resolution
- Rules are compiled
- There is a schema for the EDB facts
- Sophisticated compiler and optimizer

Launching Instructions

type:

```
~mjm/LDL/LDL++5.1
```

Note that, once at the LDL command line, you may list and change directories. In fact all the following unix commands may be issued:

```
pwd
ls
cd
popd
pushd
more
unix <command args>
```

Other commands available

While we are at it, we also have the old favorites:

```
!!
! [<pattern> | <history_index>]
history
alias
unalias
```

And we have some special commands:

```
edit <filename>
unix <command args>
set
```


LDL Commands

So far LDL is just a 'shell'. Here are the actual LDL commands:

```
open <ldl_shema_and/or_rules_file>
close
initdb <ldl_facts_filename>
savedb <ldl_database_filename>
getdb <ldl_database_filename>
compile [<exported_query_forms>]
query <predicate>
insert <ground_predicate>
delete <predicate>
display [schema|rules|facts|export]
```


Schema Definition

Define the schema in a separate file.

In this case `family.ldl` contains:

```
database({
  person(Number:integer,
         FName:string,
         LName:string,
         Gender:string),
  parent(Parent:integer,Child:integer)
}).
```

Issue the command: `load family.ldl`

EDB relations held in ASCII file

The file `bush.facts` contains the family info on the Bushes.

```
person(001,'James','Bush','M').
person(002,'Sam', 'Bush','M').
person(003,'Prescott','Bush','M').
person(004,'George','Bush','M').
person(005,'George','Bush','M').
person(006,'Jeb','Bush','M').
person(007,'Barbara','Pierce','F').

parent(001,002).
parent(002,003).
parent(003,004).
parent(004,005).
parent(004,006).
parent(007,005).
parent(007,006).
```

Just type: `initdb bush.facts`

Compiling Queries

Prepare queries with command `compile`

```
compile person(X,Y,Z,$G)
```

The allows queries like:

```
query person(X,Y,Z,'F')
```

One may also include an `export` in the the data definition file.

```
export person(X,Y,Z,$G)
```

The command `compile` (with no arguments) will compile all exported query expression.

Rules

Place rule definitions in the schema definition file.

```
father(X,Y) :- parent(X,Y),person(X,_,_,'M').
export father(X,Y).
```

And recursive rules too:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).
export ancestor(X,$Y).
export anscestor($X,Y).
export ancestor(X,Y).
```


More ...

```
assetsInCrowns(X,Y,W) :- asset(X,Y,Z), W = Z*9.7.
```

Safety is based on exports

```
good_salary(X) :- X > 70000
export good_salary($X).
export good_salary(X).
```

The first export compiles. The second produces an error.

We have some built in aggregates.

```
totalvalue(X,sum<Z>) :- asset(X,Y,Z).
```

Also `avg`,`count`,`min`,`max`,`sum`.

Negation

```
total(X,0) :- person(X,_,_,_),~asset(X,_,_).
```

Notice how this additional definition gives us a complete definition of total value.

Lets go further with negation - and exploit the statified model in LDL.

```
wealthy(X) :- total(X,V),V>10000.
wealthy(X) :- anscestor(Y,X), wealthy(Y).
send_invest_advert(X) :- wealthy(X).
send_credit_advert(W) :- person(W,X,Y,Z),
                         ~wealthy(W).
```

Becareful, because the appearance of variables in a negated predicated does not limit that variable.

A quirk:

```
send_credit_advert(W) :- ~wealthy(W),
                         person(W,X,Y,Z).
```

Does not work. Predicate ordering important.

Existential Variables

Existential Variables appear only once in a rule.

They are mean 'there exists' when used in positive literals

They mean 'there does not exixt' when used in negative literals

```
poor(X) :- totalvalue(X,V),V < -10000.
print_status('every one is doing OK') :- ~poor(Y).
```

Note that print status is equivalent to:

```
print_status('every one is doing OK') :-
                                ~someonePoor.
someonePoor :- poor(Y).
```


Stratification

No recursion through a negation.

```
even(0).
even(Y) :- ~even(X),Y=X+1.
```

The way to this is rather:

```
int(1).
int(Y) :- int(X), Y=X+1.
odd(X) :- int(X), X mod 2 ~= 0.
even(Y) :- int(X),~odd(X).
```

Which exports over `even` and `odd` will compile?

Complex Terms

Complex terms in LDL may occupy any argument position and consist of a functor and several arguments (which may also be complex).

```
part_weight(No,Kilos) :- part(No,_,actualkg(kilos))
part_weight(No,Kilos) :- part(No, Shape, unitkg(K)),
                         area(Sgape,A),kilos= K*A.
area(circle(Dmtr),A1) :- A = Dmtr *Dmtr * 3.14/4.
area(rectangle(Base,Height),A1) :- A1=Base*Height.
```

And given the facts:

```
part(322, circle(11), actualkg(34)).
part(121, rectangle(10,10), unitkg(2.1)).
```

Through unification, we are able to computer their weights!

Lists

We support lists with a special notation.

`[]` denotes the empty list

`[X|Y]` is a list with the head `X` and the tail `Y`.

`[a,b,c,...]` is short for `[a|[b|[c...]]]`

```
part('socks', [brown, black, blue]).
```

To create the unnested relation `part_color(I,C)`

```
subc(I,L) :- part(I, L).
subc(I,T) :- subc(I, [H|T]).
part_color(I,C) :- subc(I, [C|_]).
```

Sets

```
armed('Zulus', {sticks, stones, spears}).
armed('Indians',{arrow, spears}).
armed('Pilgrim',{musket,sword}).
armed('Pirates',{musket,sword}).
```

Notice that order is not important.

```
query armed(X,{sword,musket})
```

Nor is duplication:

```
query armed(X,{sword,musket,sword})
```

Here we show the use of a built in set predicate

```
has_arms(X,A) :- armed(X,S), member(A,S).
query has_arms(X,'musket')
```

we also have: subset, union, difference, intersection, and cardinality.

Aggregates

We can apply aggregate operators to sets:

```
number_of_weapon_types(X, No) :-
  armed(X,S),aggr(count,S,No).
```

We can group sets together through expressions such as

```
sup_parts(Sup, <part>) :-
  supp(Sup,Part, Price), Price > 5.00.
```


Choice

In many practical situations, it is necessary to (nondeterministically) choose just one of the elements in a set.

```
student('Mike M.',cs, grad).
professor('Chu',cs).
professor('Parker',cs).
professor('Zaniolo',cs).
professor('Stabler',linguistics).
```

But now we need to assign the student exactly one advisor.

```
advisor(S,P) :- student(S, Major, Rank),
                professor(P,Major), choice((S),(P)).
```

Non-detrministically makes the choice.

Note how this enforces the functional dependency that student determines professor.

Updates

The + and - operators insert and delete facts from the database.

Update constructs depend on predicate ordering. The first predicates are the query part, the second list of predicates are the update part.

```
delete_small_cities(Size) :-
     city(Name, State, Population),
     Population < Size,
     -city(Name, State, Population).
```

updates may only be applied to base relations.

No recursion

No union rules

Conclusions

Showed the important constructs (for this class) of LDL.

Important Points.

- LDL is a real system
- Experiment with LDL++ to get a feeling for it
- Delarative programming may be very elegant, or very confused and slow.
- Just like all programming, it is part art.