memo.pl

Tim Menzies
Lane Deptartment of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV 26506
tim@menzies.us

Contents

1 Memos

Memoing with inconsistency checking.

Each memo is a pair `Key=Value`. If the program generates a completely new `Value` for `Key`, then it is stored.

If the program stumbles on another `Value` for `Key`, then this code will reject the new `Value` if contradicts the older known `Value`.

At anytime, the program can ask for the current `value`.

Internally, the `Key=Value` pairs are stored as `assumption`s since if the program backtracks over memo creation, the `Value` is thrown away. That is, all memos are tentative and can be discarded if that proves useful.

2 Header

2.1 Loads */

```
:- [ecg] . /*
```

2.2 Flags */

```
:- dynamic       assumption/4.
:- dynamic       keyValue/3.
:- discontiguous keyValue/3.
:- multifile     keyValue/3.
:- index(assumption(1,1,1,0)). /*
```

3 Body

3.1 Assumptions

Assumptions are stored as `assumption(Hash,Key,Value,How)` where:

`Hash`
: Allows for fast access to `assumption`s with complex terms for `Key`.

`Key`
: The *name* of the thing assumed.

`Value`
: The *value* of the thing assumed.

`How`
: Some comment on how we got to this `assumption`.

*/

```
reset :- retractall(assumption(_,_,_,_)). /*
```

3.2 keyValue

In order to allow assumptions on arbitrary terms, the keyValue(Term,Index,Key,Value) predicate inputs some Term and pulls it apart into its Key and Value. Then it hashes on the Key to find the Index. */

```
keyValue(Term,Hash,Key,Value) :-
   once(keyValue(Term,Key,Value)), hash_term(Key,Hash). /*
```

To customize its behavior, add keyValue/3 facts: */

```
keyValue(Key=Value,Key,Value).
keyValue(Term,Term,t). /*
```

3.3 Assume

With all that defined, now we can assume things: */

```
assume(X,_)   :- keyValue(X,H,In,Out), assumption(H,In,Old,_),!,Out=Old.
assume(X,How) :- keyValue(X,H,In,Out), bassert(assumption(H,In,Out,How)). /*
```

3.4 Memo

That's all under-the-hood stuff. The main driver of the memo system is memo(Goal,Results).
*/

```
memo(Goal,Memos) :-
   status(Memos,New),
   (New=0 -> true; Goal, ok(Memos,byRule)).

def(status,[contradictions,agreements, new]).

status(Memos,New) :- status(Memos,New,_,_).

status(L,Flag) --> in status,
   statusReset,
   statusRun(L),
   the new=Flag.

statusReset--> in status, the contradictions:=0, the agreements:=0, the new:=0.

statusRun([]) -->[].
statusRun([Term|Terms]) -->  in status,
   {keyValue(Term,Hash,Key,Value)},
   statusStep(Hash,Key,Value),
   the contradictions < 1,
   statusRun(Terms). /*
```

Three cases:

1. 1

   Old Key is missing: we have a new key to find. */

```
    statusStep(H,K,_) --> in status, {\+ assumption(H,K,_,_)   },  +the new. /*
```

2. 1

   Old `Key` is present, new value is bound: check that old=new, otherwise we should note a contradiction. */

```
    statusStep(H,K,V) --> in status, {ground(V), assumption(H,K,V,_ )},          +th
    statusStep(H,K,V) --> in status, {ground(V), assumption(H,K,V0,_), V0 \= V}, +th
```

3. 1

   Old `Key` is present, new value is unbound: bind new value to old value.*/

```
    statusStep(H,K,V) --> in status, {var(V), assumption(H,K,V,_)}  , +the agreement
```

```
*/

ok([],_).
ok([H|T],How) :- assume(H,How),ok(T,How). /*
```

3.5 Utils

Ye olde backtrackable assert. Good for recording information about assumptions, then forgetting about them if that don't pan out. */

```
bassert(X) :- assert(X).
bassert(X) :- retract(X),fail.
```