gv.pro: Global Variables in Prolog
A User's Guide

Jacob Peck
State University of New York, College at Oswego

September 14, 2011

Abstract

This document outlines the implementation and use of pseudo-global variables in Prolog through the use of the `gv.pro` library by Craig Graci. Explanation of various functionality and example cases for each construct are given, along with technical details on implementation where appropriate.

Contents

1 Introduction

Global variables are a very powerful construct in many sequential programming languages, and are utilized to provide complex functionality with relative simplicity compared to global-free code. Though the use of global variables is oftentimes not considered "best practice" in coding, in many ways they are vital to writing intelligible implementations of algorithms quickly and easily. It seems, then, that any sequential language that wishes to allow ease of expressivity as well as prevent cognitive disequilibrium should implement global variables in some way that is idiomatic to programming.

Prolog is an example of a sequential programming language that has no notion of a global variable. In Prolog, all variables are only in scope for the query in which they operate, which causes some difficulty when attempting to port a sequential algorithm to Prolog, or indeed when attempting any sort of computation that requires more than a single query.

Through manipulation of Prolog's featured data type (the relation), `gv.pro` allows one to declare pseudo-global variables in order to store data between queries.

2 Implementation

The `gv.pro` library stores bindings in a relation called `binding` of degree two, with the first atom being a symbolic atom representing the name of a variable, and the second atom being whatever the value of that variable should be. Three things are of note here: *a*) while Prolog variable names must start with an uppercase letter, variable names within the scope of `gv.pro` must consist completely of lowercase letters so as to be a symbolic atom; *b*) there is absolutely no restriction whatsoever on the contents of any variable defined by `gv.pro`, as long as those contents are valid in relation to Prolog itself; and *c*) there is absolutely nothing keeping the end user from accessing this `binding` relation, and as such variables and their bindings may be processed just as any other degree-two relation in Prolog.

3 Syntax Descriptions

What follows is a description of all of the syntax related to the `gv.pro` library. Any examples presented use output from SWI-Prolog, and any line prefixed with `?-` represents the user-supplied command. All examples can be considered to work off of each other in sequential order.

3.1 Declaring a variable

3.1.1 High-level description

To declare a variable, use the `declare(Variable,Value)` function, where `Variable` is the symbolic atom representing your variable name, and `Value` is the initial value of the variable.

When calling `declare`, `gv.pro` checks for the existence of a variable with the name `Variable`, and if discovered, overwrites it with the value `Value`. If there is no preexisting variable `Variable`, a new one is created with the initial value `Value`.

3.1.2 Example usage

```
?- declare(x,42).
Yes

?- declare(whoami,deepthought).
```

```
Yes

?- declare(dontforget,'to bring a towel!').
Yes
```

3.2 Rebinding a variable

3.2.1 High-level description

To rebind a variable that already exists, use the `bind(Variable,Value)` function, where `Variable` and `Value` hold the same meanings as when declaring a variable.

When using `bind` instead of `declare`, if `Variable` does not exist, it is not created; rather, Prolog considers it a failure and returns `No`. Otherwise, `bind` is identical to `declare`.

3.2.2 Example usage

```
?- bind(x,23).
Yes

?- bind(somevariablethatdoesntexistyet,blah).
No
```

3.3 Undeclaring a variable

3.3.1 High-level description

To undeclare a variable, use `undeclare(Variable)`, where `Variable` is the name of a variable that is already bound.

When `undeclare` is called, any reference to `Variable` is erased from the `binding` relationship, returning `Yes` in case of successful removal and `No` in the case of no reference found.

3.3.2 Example usage

```
?- undeclare(dontforget).
Yes

?- undeclare(somevariablethatdoesntexistyet).
No
```

3.4 Retrieving the value of a variable

3.4.1 High-level description

To retrieve the value of a variable, use the `valueOf(Variable,Placeholder)` function, which places the value bound to `Variable` into the Prolog variable `Placeholder` for the scope of the query.

When `valueOf` is called on an undeclared variable, Prolog's pattern matcher fails and returns `No`.

3.4.2 Example usage

```
?- valueOf(whoami,Who).
Who = deepthought
Yes
```

```
?- valueOf(somevariablethatdoesntexistyet,Blah).
No
```

3.5 Incrementing a variable

3.5.1 High-level description

To increment a variable that holds a numerical value, use the `inc(Variable)` function, where `Variable` is the variable name.

When called on a non-numerical value, `inc` throws an exception (but only *after* unbinding `Variable`! This is unsafe, so use with care). When called on a non-existent variable, `inc` fails and returns `No`.

3.5.2 Example usage

```
?- inc(x).
Yes

?- inc(whoami).
ERROR: is/2: Arithmetic: ‘deepthought/0’ is not a function
^  Exception: (8) _L161 is deepthought+1 ? abort
% Execution Aborted

?- inc(somevariablethatdoesntexistyet).
No
```

3.6 Decrementing a variable

To decrement a variable that holds a numerical value, use the `dec(Variable)` function, where `Variable` is the variable name.

When called on a non-numerical value, `dec` throws an exception (but only *after* unbinding `Variable`! This is unsafe, so use with care). When called on a non-existent variable, `dec` fails and returns `No`.

3.6.1 High-level description

3.6.2 Example usage

```
?- dec(x).
Yes

?- declare(whoami,deepthought).
Yes

?- dec(whoami).
ERROR: is/2: Arithmetic: ‘deepthought/0’ is not a function
^  Exception: (9) _L161 is deepthought-1 ? abort
% Execution Aborted

?- dec(somevariablethatdoesntexistyet).
No
```

3.7 Adding a value to a variable

3.7.1 High-level description

To add a value to a variable's stored value and bind that variable to the new value, use the `add(Variable,Value)` function, where `Variable` is a variable name and `Value` is the value to add to the stored value assigned to `Variable`.

When called on a non-numerical value, `add` throws an exception (but only *after* unbinding `Variable`! This is unsafe, so use with care). The same results when `Value` is non-numerical. When `Variable` is unbound, `add` fails and returns `No`.

3.7.2 Example usage

```
?- add(x,30).
Yes

?- declare(whoami,deepthought).
Yes

?- add(whoami,40).
ERROR: is/2: Arithmetic: 'deepthought/0' is not a function
^  Exception: (9) _L162 is deepthought+40 ? abort
% Execution Aborted

?- declare(y,119.4).
Yes

?- add(y,'I\'m a cat.').
ERROR: is/2: Arithmetic: 'I'm a cat. /0' is not a function
^  Exception: (9) _L162 is 119.4+'I\'m a cat.' ? abort
% Execution Aborted

?- add(somevariablethatdoesntexistyet,3).
No
```

3.8 Displaying all current bindings

3.8.1 High-level description

To display all bindings currently employed by `gv.pro`, use the `displayBindings` function. This function takes no parameters and simply prints out a list of the variables tracked by `gv.pro` one to a line, in the format `<variable name> -> <variable value>`.

3.8.2 Example usage

```
?- displayBindings.
x -> 53
Yes

?- declare(somevariablethatdoesntexistyet,'Finally!, I exist! :D').
Yes
```

```
?- displayBindings.
x -> 53
somevariablethatdoesntexistyet -> Finally!, I exist! :D
Yes

?- undeclare(x).
Yes

?- displayBindings.
somevariablethatdoesntexistyet -> Finally!, I exist! :D
Yes

?- undeclare(somevariablethatdoesntexistyet).
Yes

?- displayBindings.
Yes
```

4 Listing of gv.pro

```
% FILE: gv.pro
% TYPE: Prolog source
% LINE: very simple global variable ADT
% DATE: November, 1995

  declare(Var,Val) :-
    retract(binding(Var,_)),
    assert(binding(Var,Val)).
  declare(Var,Val) :-
    assert(binding(Var,Val)).

  bind(Variable,Value) :-
    retract(binding(Variable,_)),
    assert(binding(Variable,Value)).

  valueOf(Variable,Value) :-
    binding(Variable,Value).

  undeclare(Var) :-
    retract(binding(Var,_)).

  inc(Variable) :-
    retract(binding(Variable,Value)),
    NewValue is Value + 1,
    assert(binding(Variable,NewValue)).

  dec(Variable) :-
    retract(binding(Variable,Value)),
```

```
29      NewValue is Value - 1,
30      assert(binding(Variable,NewValue)).
31
32    add(Variable,Number) :-
33      retract(binding(Variable,Value)),
34      NewValue is Value + Number,
35      assert(binding(Variable,NewValue)).
36
37    displayBindings :-
38      binding(Variable,Value),
39      write(Variable),write(' -> '),write(Value),nl,
40      fail.
41    displayBindings.
```