

Interclausal Logic Variables

PAUL TARAU and FAHMIDA HAMID

Department of Computer Science and Engineering

(e-mail: tarau@cs.unt.edu) (e-mail: fahmidahamid@my.unt.edu)

submitted February 14, 2014; revised April 15, 2014; accepted March 25, 2014

Abstract

Unification of logic variables instantly connects present and future observations of their value, independently of their location in the data areas of the runtime system. The paper extends this property to “interclausal logic variables”, an easy to implement Prolog extension that supports instant global information exchanges without dynamic database updates. We illustrate their usefulness with two of algorithms, *graph coloring* and *minimum spanning tree*. Implementations of interclausal variables as source-level transformations and as abstract machine adaptations are given. To address the need for globally visible chained transitions of logic variables we describe a DCG-based program transformation that extends the functionality of interclausal variables.

KEYWORDS: declarative programming language constructs, Prolog implementation, logic variables, definite clause grammars, continuation passing Prolog.

1 Introduction

Referred to by Einstein as “spooky action at a distance”, quantum entanglement is the fact that observation of the values of a particle’s physical attributes binds instantly entangled particles to identical values independently of their physical distance.

In the field of quantum computing, entanglement plays a crucial role in designing new algorithms and communication mechanisms, as well as in fine-tuning physical realizations of quantum computing machines (Panangaden 2011).

In logic programming languages, the prototypical instance of such an “entanglement pattern” is unification of logic variables (Robinson 1965). It instantly connects present and future observations of their value, independently of their location in the data areas of the runtime system.

While indulging into deviations from the strict entanglement analogy, thinking in terms of it can clarify some interesting algorithms that are part of the “folklore” of logic programming since the early years of Prolog. For instance, a simple and elegant graph coloring algorithm is derived by using logic variables to denote colors associated to a vertex. Avoiding cycles in graph visiting algorithms, solving a knight’s tour puzzle or finding a Hamiltonian circuit in a graph have also simple declarative programs exhibiting the entanglement analogy centered on unique bindings to logic variables.

We will revisit a few of these algorithms while proposing some new language constructs. The fact that they are unusually easy to implement in a language like Prolog, gives us hope that they will lead to interesting uses in everyday programming.

The paper is organized as follows. Section 2 introduces interclausal variables. Section 4 describes their implementation in the Styla Prolog system and discusses some alternative source-level and WAM-level implementations. Section 3 describes the use of interclausal variables in algorithms like graph coloring and minimum spanning tree and their use to inject dynamic code in a program without using assert operations. Section 5 discusses a source-level implementation of backtrackable assumptions using Prolog’s DCG transformation. Section 6 discusses related work and section 7 concludes the paper.

2 Interclausal logic variables

A natural extension of unification seen as an instance of the entanglement pattern is to apply it to variables shared among different clauses, that we will call here *interclausal variables*.

In a given logic program we could syntactically mark such a variable X , shared among clauses, as $\sim X$, for example

```
a( $\sim X$ ).  
b( $\sim X$ ).
```

The execution algorithm will then be modified to share bindings between $\sim X$ occurring in the two clauses as in

```
?- a(10),b(V).  
a(10),b(V).  
V = 10.
```

At the same time, it makes sense to trail such bindings, as one would do with ordinary logic variables. This means that a query like the following would also succeed, with a different binding

```
?- a(V),b(20).  
a(V),b(20).  
V = 20.
```

Therefore, the semantics of interclausal variables is the same as passing them along in a shared compound term containing them as arguments, or passing them directly as additional arguments to all predicates occurring in the program. Like in the case of ordinary logic variables, their behavior on backtracking provides a form of memory reuse. At the same time, indexing of Prolog clauses provides comparable access to the shared variables as if they would be passed along in a data structure or as extra arguments.

As a result of our intended semantics, one would also expect that the interclausal variables are trailed and reset to free after the query is answered.

3 Interclausal Variables at Work

Interclausal variables can be used in Prolog facts representing (possibly large) graphs as markers associated to vertices. This assertional representation can provide scalability and memory efficiency superior to equivalent representations as a data structure.

3.1 Graph Coloring with Interclausal Variables

We will start by illustrating a use of interclausal variables on a graph coloring program, derived from a classic example exhibiting the use of logic variables as colors to be assigned to vertices.

First we define our colors:

```
color(red). color(green). color(blue).
```

Next we define our vertices with interclausal variables $\sim C1.. \sim C6$ representing the colors associated to each vertex.

```
vertex(1,~C1). vertex(2,~C2). vertex(3,~C3).
vertex(4,~C4). vertex(5,~C5). vertex(6,~C6).
```

The graph will be described as a set of edges connecting our vertices.

```
edge(1,2). edge(2,3). edge(1,3). edge(3,4). edge(4,5).
edge(5,6). edge(4,6). edge(2,5). edge(1,6).
```

The coloring algorithm will iterate over all edges to color their endpoint vertices and then collect the facts describing the colorings.

```
coloring(Vs):-
  E=edge(_,_),findall(E,E,Es),
  color_all(Es),
  V=vertex(_,_),findall(V,V,Vs).
```

The iteration over all edges ensures at each step that adjacent vertices are colored differently

```
color_all([]).
color_all([edge(X,Y)|Es]):-
  vertex(X,C), color(C),
  vertex(Y,D), color(D),
  \+(C=D),
  color_all(Es).
```

The algorithm will return multiple possible colorings on backtracking, as if the colors were passed along as additional arguments to each clause.

```
?- coloring(Vs).
Vs = [vertex(1,red),vertex(2,green),vertex(3,blue),
      vertex(4,red),vertex(5,blue),vertex(6,green)];
...
Vs = [vertex(1,blue),vertex(2,green),vertex(3,red),
      vertex(4,blue),vertex(5,red),vertex(6,green)].
```

At the end, the interclausal variables are ready for being reused, back to an unbound state:

```
?- listing(vertex).
vertex(1,~C1).
...
vertex(6,~C6).
```

Note the mild deviation from our entanglement analogy, given that (sound) negation as failure is used to ensure that colors associated to neighboring vertices are distinct.

Note also that in ASP systems (Gebser et al. 2007) or SAT-based constraint solver extensions to Prolog (Zhou 2013) that rely on grounding, interclausal variables could be introduced with the same semantics, to control combinatorial explosion that depends on the total number of distinct variables.

3.2 A Minimum Spanning Tree Algorithm using Interclausal Variables

Our next example uses interclausal variables for a variant of Kruskal’s minimum spanning tree algorithm with logic variables working as markers for connected sets of edges that grow progressively until they cover the graph (assuming it is connected). It has been derived from a Prolog program using a data structure passed along between clauses and posted on Usenet by the author in 1992¹.

The algorithm proceeds by first sorting by cost the set of edges.

```
mst(NbOfVertices,Edges,MinSpanTree):-
    sort(Edges,SortedEdges),
    mst0(NbOfVertices,SortedEdges,MinSpanTree).
```

Next the program explores the set of edges, given as the second argument of the predicate `mst0/3`. At a given step, it calls the predicate `mst1/7` which decides about unifying or not the components `C1` and `C2`.

```
mst0(1,_,[]). % no more vertices left
mst0(N,[E|Es],T):- N>1,
    E=edge(_Cost,V1,V2),
    vertex(V1,C1), % C1,C2 are the components of V1,V2
    vertex(V2,C2),
    mst1(C1,C2,E,T,NewT,N,NewN),
    mst0(NewN,Es,NewT).
```

The predicate `mst1/7` checks if both endpoints of an edge are already in an incrementally grown set of connected edges, in which case it skips the edge. Otherwise, if the sets represented by `C1` and `C2` are distinct, they will be merged by unifying the variables, adding the edge to the minimum spanning tree and counting the vertex as processed. Note that we are reusing here the vertex definitions of our graph coloring program, with colors interpreted as components.

```
mst1(C1,C2,_,T,T,N,N):-C1==C2.
mst1(C1,C2,E,T,NewT,N,NewN):-C1\==C2,C1=C2,
    % Put endpoints in the same component
    T=[E|NewT], % Add the the edge to the MST
    NewN is N-1. % Count a new vertex
```

Finally the predicate `test_mst` tries out the algorithm on a small graph.

```
test_mst(MinSpanTree):-
    Edges = [ edge(70,1,3),edge(80,3,4),edge(90,1,5),
               edge(60,2,3),edge(20,4,5),edge(30,1,4),
               edge(40,2,5),edge(50,3,5),edge(10,1,2)
             ],
    mst(5,Edges,MinSpanTree).
```

¹ A time when such uses of logic variables were still waiting to be uncovered.

Note that an answer is returned as a list of edges ordered by cost, such that each vertex is an endpoint of at least one edge.

```
?- test_mst(Mst).
Mst = [edge(10,1,2),edge(20,4,5),edge(30,1,4),edge(50,3,5)]
```

3.3 Injecting Dynamic Code without Asserts

When used in a metavariable position, an interclausal variable can provide a lightweight alternative to the `assert/retract` interface to dynamic code. In a clause like $a_0:-a_1,\dots,\sim V,\dots,a_n$ the metavariable $\sim V$ can be bound to a Prolog terms that gets “injected” in the possibly statically compiled code of the clause. In particular, injecting $\sim V=\text{fail}$ in a clause like $a_0:-\sim V,\dots,a_n$ would temporarily disable the clause without the need to use a `retract` operation. In a different branch of the computation, one could inject $\sim V=\text{true}$ to enable the clause.

4 Implementing Interclausal Variables

We will describe here a few mechanisms for adding support for interclausal variables to Prolog systems.

4.1 Interclausal variables in *Styla*

We have implemented *interclausal variables* in our *Styla* Scala-based Prolog system (Tarau 2012a) by taking advantage of its object oriented term structure and its distributed unification and term copying algorithms, designed in such a way that various subterms contribute small steps depending on their type. We have also used the fact that inheritance enables “surgical” overriding of the small methods implementing these algorithms together.

First, we have created a new type `EVar` for interclausal variables as an extension of the class of ordinary logic variables `Var`.

```
package prolog.terms
class EVar() extends Var {
  override def tcopy(dict : Copier) : Term = this.ref
}
```

We made it inherit all properties of logic variables except one: behavior on copying. The method `tcopy`, instead of creating a fresh variable, simply returns the reference `ref` of our interclausal variable. As a result, bindings of interclausal variables are shared between calls, while their unification behavior, including trailing for undoing bindings on backtracking, is inherited unchanged.

Styla uses Scala’s *combinator parsing* API where only two simple modifications were needed to process our new data type.

First, we specified the regular expression

```
val evarToken: Parser[String] = ""~"[A-Z_]\\w*""~".r
```

defining that interclausal variables start with the $\sim$ symbol and have, otherwise, the same token specification as the usual ones.

Next, we have ensured that the parser knows about them, by adding a rule associated to their token type calling the method `mkEVar`

```
def mkEVar(x: String) = {
    vars.getOrElseUpdate(x, new EVar())
}
```

Finally, a small change to the `toString` method marks with a “~” the string representation of interclausal variables. Besides helping with debugging, this is also useful as *Styla* keeps track of variable names in the source code and uses them in predicates like `listing/1`, when the source code of a predicate is displayed.

4.2 Source-level Implementations

Given a set of interclausal variables, one can implement them at source level simply by adding them as extra arguments to each clause of a program. This would ensure that a Datalog program remains a Datalog program after the transformation. While linear, the resulting code explosion can be avoided by adding a single variable to each clause representing a compound term, together with an `arg/3` predicate call accessing the appropriate position in it, for each interclausal variable occurring in a given clause.

4.3 WAM-level Implementations

In a way similar to *BinProlog*’s implementation of multiple DCG streams (Dahl et al. 1997), the argument registers (represented as an array in *BinProlog* (Tarau 2012b)) can be extended with as many positions as needed to accommodate all interclausal variables, to which the compiler would generate appropriate references in instructions like `unify_variable` and `unify_value` (Aït-Kaci 1991). Alternatively, a heap area could be reserved for them, say at a lower address range than that reserved for ordinary variables, and instructions would be generated to create them on the heap before execution begins.

4.4 Scoping constructs and interclausal logic variables

Limiting the scope of interclausal variables to smaller code units can be achieved easily in the case of a source-level implementation by limiting their addition as extra arguments to only the clauses of a given module.

On the other hand, in a Prolog systems that would support *local clauses*, with a semantics similar to Haskell’s “**where**” construct (usable for local function definitions), one could implement variants of interclausal variables as logic variables one or more levels up from the point where they are used with or without copying on new clause calls.

5 Source-level backtrackable assumptions

We will overview here another, less “pure” instance of the entanglement pattern that provides, at source-level, a richer set of functionalities than interclausal or backtrackable global variables.

A limitation of interclausal variables is that they do not allow threading information that changes over multiple recursive calls, for which the prototypical example is Prolog’s

Definite Clause Grammar (DCG) mechanism (Pereira and Warren 1980), which has been extended to support multiple independent chains of variables at source level (Van Roy 1989) or at WAM-level (Tarau et al. 1995).

As an application of the WAM-level implementation of (Tarau et al. 1995), specific to the BinProlog system, *Assumption Grammars* have been introduced in (Dahl et al. 1997) featuring backtrackable dynamic database updates and a mechanism allowing the programmer to chose between copying or sharing semantics for the assumed clauses.

We will describe here a source level implementation of the functionality of Assumption Grammars, by overloading the standard DCG mechanism. As a result, it is portable to virtually all Prolog systems.

Note that predicates defined here with arity 3 should be used within clauses defined with DCG arrow “`-->/2`” rather than the usual clause neck “`:-/2`”.

The Assumption Grammar API is implemented as follows as source level program transformation.

5.1 Setting and getting the database and the DCG tokens

`'#<' (Xs)` sets the DCG token list to be `Xs` for processing by the assumption grammar.

```
'#<' (Xs,_,Db-Xs):-new_assumption_db(Db).
```

`'#>' (Xs)` unifies current assumption grammar token list with `Xs`.

```
'#>' (Xs,Db-Xs,Db-Xs).
```

`'#:' (X)` matches `X` against the current DCG token the assumption grammar is working on.

```
'#:' (X,Db-[X|Xs],Db-Xs).
```

5.2 Adding new assumptions

`'#+' (X)` adds “linear” assumption `+(X)` to be consumed at most once, by a `'#-'` operation.

```
'#+' (X,Db1-Xs,Db2-Xs):-add_assumption('+'(X),Db1,Db2).
```

Note that variables occurring in a clause assumed with the `'#+'` operation are “inter-clausal” and their bindings provide a long distance communication channel between the points where they are produced and consumed. `'##' (X)` adds ‘intuitionistic’ assumption `'*' (X)` to be used indefinitely by `'#-'` operation.

```
'##' (X,Db1-Xs,Db2-Xs):-add_assumption('*'(X),Db1,Db2).
```

The semantics of these clauses is essentially the same as Prolog’s dynamic database with “immediate update”, except that assumptions are backtrackable.

5.3 Querying the assumptions

`'#=' (X)` unifies `X` with any matching existing or future `'+' (X)` linear assumptions.

```
'#=' (X,Db1-Xs,Db2-Xs):-equate_assumption('+'(X),Db1,Db2).
```

'#-'(X) consumes a +(X) linear assumption or matches a '*'(X) intuitionistic assumption.

```
'#-'(X,Db1-Xs,Db2-Xs):-consume_assumption('+'(X),Db1,Db2).
'#-'(X,Db-Xs,Db-Xs):-match_assumption('*'(X),Db).
```

Note that this operation provides a mechanism to call either linear or intuitionistic assumptions, except that in the later case, matching assumptions are “consumed” i.e; removed from the database. '#?'(X) matches '+'(X) or '*'(X) assumptions without any binding.

```
'#?'(X,Db-Xs,Db-Xs):-match_assumption('+'(X),Db).
'#?'(X,Db-Xs,Db-Xs):-match_assumption('*'(X),Db).
```

5.4 Auxiliary predicates

A few auxiliary predicates implement internals of the API:

```
new_assumption_db(Xs/Xs).

add_assumption(X,Xs/[X|Ys],Xs/Ys).

consume_assumption(X,Xs/Ys,Zs/Ys):-nonvar_select(X,Xs,Zs).

match_assumption(X,Xs/_):-nonvar_member(X0,Xs),copy_term(X0,X).

equate_assumption(X,Xs/Ys,XsZs):- \+(nonvar_member(X,Xs)),!,
    add_assumption(X,Xs/Ys,XsZs).
equate_assumption(X,Xs/Ys,Xs/Ys):-nonvar_member(X,Xs).
```

Finally, nonvar_member(X,XXs) and nonvar_select(X,XXs,Xs) are variants of member/2 and select/3 working on open ended lists.

```
nonvar_member(X,XXs):-nonvar(XXs),XXs=[X|_].
nonvar_member(X,YXs):-nonvar(YXs),YXs=[_|Xs],nonvar_member(X,Xs).

nonvar_select(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
nonvar_select(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],nonvar_select(X,Xs,Ys).
```

5.5 Using Assumption Grammars

One can use phrase/3 to test out assumption grammar components, as follows:

```
?- phrase(('#<'([a,b,c]),'+'(t(99)),'*'(p(88)),'#-'(t(A)),'#-'(p(B)),
    '#:'(X),'#>'(As)),Xs,Ys).
A = 99, B = 88, X = a, As = [b, c],
Ys = [*(p(88))|_G2344]/_G2344-[b, c] .

?- phrase(('#<'([a,b,c]),'#+'(t(99)),'*'(p(88)),'#-'(t(A)),
    '#-'(p(B)),'#:'(X),'#>'(As)),Xs,Ys).
A = 99, B = 88, X = a, As = [b, c],
Ys = [*(p(88))|_G1161]/_G1161-[b, c] .
```

We refer to (Dahl et al. 1997) for various examples of their use both for expressing concisely some Prolog algorithms and for capturing long distance dependencies in natural language processing phenomena like anaphora resolution and agreement.

Note that one could also implement similar constructs by combining interclausal variables storing compound terms in which mutable backtrackable state is updated with built-ins like `setarg/3`.

6 Related Work

The first author must confess that about 25 years ago he has thought about and even wrote a short draft paper about interclausal logic variables that got forgotten and lost. Being quite sure that something similar might have popped-up over time and has made it into the logic programming folklore, we have not revisited the subject until now, except for a footnote in (Tarau and Majumdar 2009) where inter-clausal variables are mentioned as write-once global variables relating them to the semantics of term copying. Other than that, we have not found despite an extensive search, any reference to them or closely related concepts.

In (Tarau and Dahl 1994), after applying the binarization transformation (Tarau 1993), multi-headed clauses are introduced, which give direct access to continuations at source-level. The technique makes possible long distance communication between logic variables otherwise inaccessible.

Global variables (both backtrackable and persistent) have been present in BinProlog since the mid-1990s (De Bosschere and Tarau 1996) and are these days available in various Prolog systems. Among them, we mention SWI-Prolog’s implementation (Wielemaker et al. 2012) where their values live on the Prolog global stack. Like in the case of interclausal variables, this implies that lookup time is independent of the size of the term. As a result, they can efficiently store large data structures like parsed XML syntax trees or global constraint stores.

By contrast to non-backtrackable global variables, our interclausal variables are single assignment and behave similarly to ordinary logic variables. Backtrackable global variables are semantically similar to interclausal variables. However, like in BinProlog 2.0’s original implementation (Tarau 1994), they are named with constants and used through an API like SWI-Prolog’s `b_setval/2` and `b_getval/2`, requiring a hash-table look-up to find their values on the heap, while the interclausal variables in this proposal are implemented simply as a special case of logic variables resulting also in a more natural notation.

7 Conclusion

Interclausal variables extend natural properties of the usual logic variables to variables shared among clauses. Given the simplicity of their implementation, for which we have outlined a few alternative scenarios, we hope they can contribute to adding flexibility to logic programming languages while keeping intact their declarative flavor.

We have also described a source-level implementation of “assumption grammars” an extension to Prolog’s DCGs that circumvents some limitations of interclausal variables.

We plan future work on implementing interclausal variables at WAM-level and experiments with their uses in probabilistic logic programming. We also plan to work on mechanisms based on interclausal logic variables that optimize the grounding phase in ASP systems and SAT-based constraint solvers used by Prolog systems.

References

- AÏT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.
- DAHL, V., TARAU, P., AND LI, R. 1997. Assumption Grammars for Processing Natural Language. In *Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed. MIT press, 256–270.
- DE BOSSCHERE, K. AND TARAU, P. 1996. Blackboard-based Extensions in Prolog. *Software — Practice and Experience* 26, 1 (Jan.), 49–69.
- GEBSER, M., SCHAUB, T., AND THIELE, S. 2007. GrinGo: A New Grounder for Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning*, C. Baral, G. Brewka, and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer Berlin Heidelberg, 266–271.
- PANANGADEN, P. 2011. The Search for Structure in Quantum Computation. In *Foundations of Software Science and Computational Structures*, M. Hofmann, Ed. Lecture Notes in Computer Science, vol. 6604. Springer Berlin Heidelberg, 1–11.
- PEREIRA, F. AND WARREN, D. 1980. Definite Clauses for Language Analysis. *Artificial Intelligence* 13, 231–278.
- ROBINSON, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *JACM* 12, 1, 23–41.
- TARAU, P. 1993. An Efficient Specialization of the WAM for Continuation Passing Binary programs. In *Proceedings of the 1993 ILPS Conference*. MIT Press, Vancouver, Canada. poster.
- TARAU, P. 1994. BinProlog 2.20 User Guide. Tech. Rep. 94-1, Dept. d’Informatique, Université de Moncton. Feb. <ftp://clement.info.umoncton.ca/BinProlog>.
- TARAU, P. 2012a. Styla: a Lightweight Scala-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy. <https://code.google.com/p/styla/>.
- TARAU, P. 2012b. The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. *Theory and Practice of Logic Programming* 12, 1-2, 97–126.
- TARAU, P. AND DAHL, V. 1994. Logic Programming and Logic Grammars with First-order Continuations. In *Proceedings of LOPSTR’94, LNCS, Springer*. Pisa.
- TARAU, P., DAHL, V., AND FALL, A. 1995. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. In *Proceedings of ILPS’95*, J. Lloyd, Ed. Portland, Oregon, 642. poster abstract.
- TARAU, P. AND MAJUMDAR, A. 2009. Interoperating Logic Engines. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009*. Springer, LNCS 5418, Savannah, Georgia, 137–151.
- VAN ROY, P. 1989. A useful extension to Prolog’s Definite Clause Grammar notation. *SIGPLAN notices* 24, 11 (Nov.), 132–134.
- WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2, 67–96.
- ZHOU, N.-F. 2013. Picat: A scalable logic-based language and system (invited talk). In *SLATE*, J. P. Leal, R. Rocha, and A. Simões, Eds. OASICS, vol. 29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5–6.