Global Variables in Logic Programming

Peter Schachte
The University of Melbourne
Parkville, Victoria 3052, Australia
pets@cs.mu.oz.au

Abstract

We show that adding global variables to logic programming can solve some common problems of reliability and programmer productivity in large logic programs. By presenting a formal semantics for logic programs with global variables, we show that this addition retains a clean semantics. This addition has the consequences that conjunction is neither commutative nor absorptive, but we show that the practical loss is small. We also describe an implementation of Prolog with global variables as a translator to ordinary Prolog which preserves the efficiency of the Prolog program, while statically detecting some programming errors, and avoiding others altogether.

1 Introduction

In small example programs as commonly found in texts on logic programming, there is not much information that needs to be passed around, so predicates usually have few arguments, rarely more than four or five. However, in real programs there is typically much more information that needs to be passed around. Information that becomes available in one place in the call graph of the program but is needed in another place must be passed as an argument in each predicate between these two places. If information is accumulated throughout a computation, two arguments are needed: one for the information up to the call and one for the information following it.

This is why it is quite common for predicates in all but the smallest logic programs to have many arguments that are only passed on to other predicates without being directly used. Contrary to what one sees in logic programming texts, it is not unusual for real world predicates to have ten or twelve arguments. Unfortunately, this tangle of arguments causes several problems in the development and maintenance of logic programs.

Probably the most important consequence of having predicates with a dozen arguments is a greater likelihood of program errors. Very simply, the more code you write, the more likely you are to make a mistake. It is easy to switch two arguments around, or omit one, or even just misspell one. Singleton variable warnings or type checkers may warn of some of these errors, in which case the error only wastes time, but no analysis can find all errors, so some of these error may go undetected until much later.

Just having to type all of these variable names over and over and to fix the inevitable typographical errors reduces productivity. Worse, one often finds in maintaining a program that new arguments must be added to many existing predicates in order to communicate through the program information whose need was unanticipated. Adding these extra arguments is also tedious and error-prone.

Another important effect is that programs become difficult to read and understand, which further compromises reliability and productivity. The intention of the predicate may become obscured by all the arguments littering the definition. Looked at another way, the minor details in the implementation of the predicate are given equal prominence to the main design. On first reading of a predicate definition, one usually wants these details hidden.

The rest of this paper is organized as follows. Section 2 discusses previous solutions to this problem, proposes a new solution, and discusses the price of hiding information flow. Section 3 gives a formal semantics for the proposal, and section 4 describes how it was implemented as an extension to Prolog. Finally, section 5 presents our conclusions and discusses how this work might be developed further.

2 Hiding Information Flow

We begin the presentation of our proposal by discussing other approaches to solving the problems presented above. Then we present our approach, and discuss the trade-offs associated with hidden information flow in this context.

2.1 Other Approaches

There is a long history of hiding information flow in Prolog. Because Prolog has always had `assert`/1 and `retract`/1 builtins, these provided an early way of passing unseen information. Because `assert`/1 and `retract`/1 modify the running program, however, and because they make it possible to associate multiple pieces of information with a symbol (by asserting multiple clauses for the same predicate), they can be very confusing to use. Worse, because they are "non-backtrackable" (their effect is not undone on failure), they make it more difficult to give a semantics to a program. For example, it becomes important whether an update happens before or after a failure, and the order of clauses becomes important. More pragmatically, debugging such a program is difficult, because one cannot simply retry a goal to see what happens without being sure to reset any state that has been modified. Non-backtrackable update is also inherently inefficient, because data must be copied from the heap in order to survive heap reclamation on backtracking. For all these reasons, we find non-backtrackable update unacceptable for our purposes.

Definite Clause Grammars (DCGs) have been a part of most Prolog implementations for more than a decade. DCGs allow one to write attributed grammar rules directly in Prolog, producing a simple recursive descent parser. In order to implement this, predicates that implement each grammar rule take two implicit arguments to represent a segment of the string being parsed. Because of Prolog's ability to run predicates with different input-output patterns, DCGs can often be run "backwards" to compute a string from the attributes. Viewed procedurally, the computation produces the string a little at a time, hiding from the programmer the arguments holding the string being produced.

Although they were not originally designed for this, DCGs can be used to produce any data structure, not just lists; however they can only produce one data structure in a computation. Van Roy [3] developed an extension, called EDCGs, that allows any number of argument pairs to be threaded through the program. This also works as a translator, but in this case the user declares each individual accumulator, specifying the goal to execute in order to accumulate a new term in it. The user may also declare "passed terms" which are not not really accumulators, but are simply passed unchanged to predicates. The user then must declare which predicates should get which accumulators as arguments.

More recently, Tarau *et al.* [5] have proposed another generalization of DCGs, called Assumption Grammars (AGs). AGs are implemented as a direct language extension in the BinProlog implementation [4]. Also part of the same extension, and more closely comparable to the present work, are the new primitives `assumel/1` and `assumei/1` which add facts to the program that can be used once or many times, respectively. Although in both cases these facts are removed on backtracking, the semantics of `assumei/1` requires that variables in the assumption are not shared with references. In the current implementation, this means that an "assumed" atom is copied, and copied again on invocation.

2.2 Global Variables

We propose a simpler model of state in logic programming: global variables. These variables are assigned to and referenced much like global variables in imperative languages. A global variable is "created" when it is first assigned a value, and exists until the end of the computation. Its value may be accessed at any time after it is first assigned; it is an error to attempt to access it before it has been assigned.

The primary advantage of this approach is its simplicity. Most logic programmers already understand global variables from their use in other programming paradigms. We will also show that it is easy to give a formal semantics to logic programs with global variables. Furthermore, our implementation avoids the copying of BinProlog's `assumei`/1, as well as the need for extra declarations, as required by EDCGs.

In the interest of concreteness, we present the small syntactic extension we have made to Prolog to support global variables. Firstly, a global variable is specified by an atom preceded by a dollar sign (`$`), for example `$list`. Variables are assigned using the `:=` operator of the Algol language family, for example `$list := [X|$list]`. Finally, to ensure that global variables are not used where users do not expect them to be, we require that clauses that may set or use global variables use the `::-` operator instead of `:-` to separate their heads from their bodies. Thus ordinary clauses retain their usual semantics. Also, since extra arguments are added to predicates, and not to individual clauses, it is an error to use `:-` in some clauses for a given predicate and `::-` for others. Note that, unlike DCGs, unit clauses present no problems for global variables, and may be written as unit clauses, whether other clauses for that predicate use `:-` or `::-`. Since we do not allow global variables in clause heads, a predicate defined entirely by unit clauses cannot use global variables at all, so again there is no problem.

Take as an example a program to reverse a list. In an imperative language, one would use an algorithm that initialized a variable to the empty list, and then traversed the input list "pushing" these elements onto the list being accumulated. This approach works quite well in logic programming with global variables:

```
reverse(L0, L) ::-                rev1([]).
        $rev := []                rev1([H|T]) ::-
        rev1(L0),                         $rev := [H|$rev],
        L = $rev.                         rev1(T).
```

Global variables can also be used to pass around terms without accumulating anything, as implemented in EDCGs, simply by assigning them only once. Furthermore, they can pass information out of a deep computation by assigning them once in the body of a predicate and using them only after a call to that predicate. Results can be collected forward or backward, depending on the order of the assignments (e.g., exchanging the two goals in the second clause of `rev1` above would accumulate the list in forward order).

2.3 The Cost

As we shall see in section 3, with the addition of global variables, conjunction is no longer commutative or absorptive. These properties are important to the programmers and maintainers of the program, to allow them to understand it. They are also important to compilers and other program analysis and transformation tools, to allow them to reason about program behavior.

Note firstly that we implement the addition of global variables by translating the extended program into an ordinary logic program. This means that the ability of compilers and other tools to reason about these programs is preserved, as long as the translation is performed before the program is analyzed. Further, the presence of global variables does not change the reordering a compiler may be allowed to perform. This is simply a matter of the dataflow dependencies; the same dependencies exist whether they are manifest by global variables or by argument threading. Secondly, a byproduct of the translation is a list for each predicate of the global variables it may change, directly or indirectly, as well as the variables it may use, directly or indirectly. This allows the reader to trace the flow of information borne by global variables when this is important. Therefore we believe that what is gained by the introduction of global variables is much more than is lost.

3 Semantics

We now define a formal semantics for logic programs with global variables, extending the notation and terminology of Lloyd [2]. We assume familiarity with this terminology.

First we must extend the alphabet of our first order theory to include the classes of *global variables* and *operations*. Thus the alphabet now includes these eight classes: variables, global variables, constants, functions, predicates, operations, connectives, and punctuation symbols. The set of global variables, as with variables, constants, functions, and predicates will vary from program to program; the others are fixed. The only operation symbol is assignment, written `:=`.

Next we extend the definition of a *term* to include global variables:

1. A variable is a term.
2. A global variable is a term.
3. A constant is a term.
4. If f is an n-ary function and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

A *(well-formed) formula* is defined as follows:

1. If p is an n-ary predicate and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula, called an *atom*.
2. If v is a global variable and t is a term, then $v := t$ is a formula, called an *assignment goal*.
3. If F and G are formulae, then so are $\neg F, F \wedge G, F \vee G, F \rightarrow G$, and $F \leftrightarrow G$.

We call formulae defined using only rules 1 or 2 *atomic goals*.

Intuitively, an assignment goal is much like an assignment statement in a procedural programming language: the "value" of the global variable following the assignment goal is the term to be assigned. In interpreting the term to be assigned, the values of global variables preceding the assignment goal are used, so it is not possible to create a circular term using an assignment goal.

An *interpretation* of a first order language L consists of the following:

1. A non-empty set D, called the *domain* of the interpretation.

2. A mapping, called a *global variable assignment*, or *gva*, from the global variables of L to the global variable-free terms of L.

3. For each n-ary function in L, the assignment of a mapping from D^n to D.

4. For each n-ary predicate in L, the assignment of a mapping from D^n into $\{true, false\}$

A *variable assignment* wrt an interpretation I maps each variable in the language to an element of the domain of I. A *term assignment* with respect to an interpretation I and a variable assignment A maps variables according to A, global variables according to the global variable assignment of I, and terms $f(t_1, \ldots, t_n)$ to $f'(t'_1, \ldots, t'_n)$, where I maps f to f' and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$. An atom can be given a truth value according to an interpretation and a variable assignment as the obvious extension of the term assignment; this is extended to all formulae using the usual meanings of the connectives. An interpretation I is a *model* for a closed formula F iff the truth value of F wrt I is *true*.

A term is *ground* iff it contains no variables or global variables. The *Herbrand universe* U_L for a language L is the set of all ground terms that can be made of the constants and functions of L. The *Herbrand base* B_L is the set of all pairs of atoms that can be made from the predicates of L and the terms of U_L with global variable assignments that map to terms in U_L. An interpretation for L is a *Herbrand interpretation* iff its domain is U_L, it maps constants and functions in L to themselves, and it maps global variables to elements of U_L. A Herbrand interpretation is a *Herbrand model* for a closed set of formulae S if it is a model for S. It is convenient to specify a Herbrand interpretation of a language L as a subset of the Herbrand base; that is, as the set of all the pairs $\langle F, V\rangle$ such that F is to have truth value *true* when the interpretation maps the global variables according to V. We also allow ourselves to define a language L by a set of program clauses P as the sets of variables, global variables, constants, etc., appearing in P.

We define an evaluation function $E : atomic\ goal \times gva \rightarrow gva$ in terms of an auxiliary term evaluation function $T : gva \times term \rightarrow term$ as follows:

$$\begin{array}{rcl} E(F, V) & = & \textbf{if } F \text{ is an assignment goal } v{:=}t \\ & & \textbf{then } \lambda x.(\textbf{if } x = v \textbf{ then } T(t, V) \textbf{ else } V(x)) \\ & & \textbf{else } V \end{array}$$

$$
\begin{aligned}
T(t,V) \quad = \quad & \textbf{if } t \text{ is a global variable } \textbf{then } V(t) \\
& \textbf{else if } t \text{ is a term } f(t_1,\ldots,t_n) \textbf{ then } f(T(t_1,V),\ldots,T(t_1,V)) \\
& \textbf{else } t
\end{aligned}
$$

We extend E to conjunctions as follows:

$$E(F \wedge G, V) \stackrel{\text{def}}{=} E(G, E(F,V)).$$

Note that for this definition of E, it is possible that $E(F \wedge G, V) \neq E(G \wedge F, V)$, and that $E(F \wedge F, V) \neq E(F,V)$. Thus conjunction is neither commutative nor absorptive in this logic. Happily, conjunction *is* associative, since

$$
\begin{aligned}
E(F \wedge (G \wedge H), V) &= E(G \wedge H, E(F,V)) \\
&= E(H, E(G, E(F,V))) \\
&= E(H, E(F \wedge G, V)) \\
&= E((F \wedge G) \wedge H, V)
\end{aligned}
$$

Finally, we define T_P for a program P as a mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$ as follows:

$$
\begin{aligned}
T_P(I) \quad = \quad & \{\langle A, V\rangle : A \leftarrow A_1,\ldots,A_n \in P \wedge \\
& \langle A_1, V_1\rangle \in I \wedge \ldots \wedge \langle A_n, E(A_1 \wedge \ldots \wedge A_{n-1}, V)\rangle \in I\}
\end{aligned}
$$

Clearly T_P is monotonic. This allows us to define the semantics of a program with global variables as $\text{lfp}(T_P)$. Furthermore, T_P is continuous, so

$$\text{lfp}(T_P) = T_P \uparrow \omega.$$

4 An Implementation Strategy

We implement global variables in Prolog programs by translating programs with global variables into ordinary Prolog programs. This is done by adding, if necessary, one or two extra arguments to a predicate for each global variable that it uses or changes, directly or indirectly. That is, we translate a program with global variables into the Prolog program that the user would have written without this tool. Therefore, the efficiency of programs that use this tool is not compromised.

As shown by Tarau *et al.* ([5] and [4], chapter 12), this sort of extension can also be implemented by extending the Prolog system itself. For reasons of portability, we prefer the translation approach. Furthermore, the analysis required for the translation provides very useful error reporting when an uninitialized global variable may be used, as well as useful information about dataflow through global variables.

4.1 The Translation

The translation process is quite simple, much like that used for DCGs. We simply add extra arguments after the "real" predicate arguments to hold the values of the global variables on call to and exit from that predicate. By convention, we put *input arguments* (which hold the state of the global variables at the start of the goal) first, followed by the *output arguments* (which hold the value after the goal). Each of these groups is sorted alphabetically by variable name.

4.2 The Analysis

The difficult part of implementing global variables is actually the global analysis to determine which predicates need extra inputs or outputs for which global variables. We call the set of all global variables $globs$, the set of all predicates $preds$, the set of all terms $terms$, the set of all atoms $atoms$, the set of all assignment goals $assigns$, the set of all atomic goals $goals$, and the set of all ordered conjunctions of goals $conjs$. For this analysis, it is convenient to assume the program is in clausal form. We use the notation

$$\bigcup_{H::-B} X$$

to mean the union of X over all clauses in the program $H :: -B$, and similarly for intersection. The function $pred : atoms \rightarrow preds$ determines the predicate invoked by an atom. The function $args : atoms \rightarrow 2^{terms}$ provides the arguments of an atom as a set. We assume functions $appears : terms \rightarrow 2^{globs}$ and $assigns : goals \rightarrow 2^{globs}$ which determine which global variables appear anywhere in a term, and which variable is assigned by an assignment goal, respectively. It is convenient to apply $appears$ to atoms as well; in this case

$$appears(atom) \stackrel{\text{def}}{=} \bigcup_{t \in args(atom)} appears(t)$$

The analysis is similar to the conventional reaching definition and live variable dataflow analyses (see, e.g., [1]), except that in this case we are most interested in definitions reaching across predicate calls. We specify the analysis with a number of mutually recursively defined functions over predicates.

A first step is to determine which predicates may use, and which predicates may change, which global variables, directly or indirectly. We can specify this with the two functions $uses_p : preds \rightarrow 2^{globs}$ and $changes_p : preds \rightarrow 2^{globs}$, which we define in terms of two other functions $uses_g : goals \rightarrow 2^{globs}$ and $changes_g : goals \rightarrow 2^{globs}$.

$$uses_p(p) = \bigcup_{H::-G_1,\ldots,G_n} \bigcup_{1 \le i \le n} uses_g(G_i) \textbf{ where } pred(H) = p$$

$$
\begin{aligned}
uses_g(G) &= \textbf{if } G \text{ is an assignment goal } v := t \textbf{ then } appears(t) \\
&\quad\ \textbf{else } uses_p(pred(G)) \cup appears(G) \\
changes_p(p) &= \bigcup_{H::-G_1,\ldots,G_n} \ \bigcup_{1\le i\le n} changes_g(G_i) \text{ where } pred(H) = p \\
changes_g(G) &= \textbf{if } G \text{ is an assignment goal } \textbf{then } assigns(G) \\
&\quad\ \textbf{else } changes_p(pred(G))
\end{aligned}
$$

The reader may note that these functions are not well-defined, as can be seen by trying to evaluate them for any recursive predicate. For the *uses* functions, we correct this in the usual manner by defining $u_p : (preds \rightarrow 2^{globs}) \rightarrow preds \rightarrow 2^{globs}$ and $u_g : goals \times (preds \rightarrow 2^{globs}) \rightarrow 2^{globs}$ as a simple transformation of the above definitions to avoid recursion:

$$
\begin{aligned}
u_p(U)(p) &= \bigcup_{H::-G_1,\ldots,G_n} \ \bigcup_{1\le i\le n} u_g(G_i, U) \text{ where } pred(H) = p \\
u_g(G, U) &= \textbf{if } G \text{ is an assignment goal } v := t \textbf{ then } appears(t) \\
&\quad\ \textbf{else } U(pred(G)) \cup appears(G) \\
uses_p &= \mathrm{lfp}(u_p)
\end{aligned}
$$

A similar transformation can be used to define $changes_p$ properly. In the interest of simplicity and conciseness, we assume a similar approach to deriving well-defined functions from ill-defined ones in the sequel.

Our translator could simply add an extra argument for each global variable a predicate uses, and one for each it changes, but this would add more arguments than necessary. Suppose, for example, the only mention of global variable `$pi` is in the following clause:

```
area(R,A) ::-
        $pi := 3.14159,
        A is $pi * R * R.
```

There is no need here to add an input or output argument for `$pi` to `area/2`. Whether or not to add an input or output argument for a global variable depends on *demand*. Here we see that the assignment to `$pi` unambiguously defines it before it is used. Clearly any previous value for `$pi` is irrelevant, so no input argument for `$pi` is needed, and since no other predicate uses `$pi`, no output is needed, either.

Let us formalize the concept of *unambiguous definition* by defining the functions $defines_p : preds \rightarrow 2^{globs}$, $defines_c : conjs \rightarrow 2^{globs}$, and $defines_g : goals \rightarrow 2^{globs}$.

$$
defines_p(p) = \bigcap_{H::-B} defines_c(B) \textbf{ where } pred(H) = p
$$

$$
\begin{aligned}
\mathit{defines}_c(B) &= \mathit{defines}_g(G_1) \cup (\mathit{defines}_c(G_2, \ldots, G_n) \setminus \mathit{uses}_g(G_1)) \\
& \;\textbf{where } B = G_1, \ldots, G_n
\end{aligned}
$$

$$
\begin{aligned}
\mathit{defines}_g(G) &= \textbf{if } G \text{ is an assignment goal } v := t \textbf{ then } \{v\} \setminus \mathit{appears}(t) \\
& \;\textbf{else } \mathit{defines}_p(\mathit{pred}(G)) \setminus \mathit{appears}(G)
\end{aligned}
$$

Thus, a predicate unambiguously defines a variable if it always changes it, and never uses it before changing it.

Note here that when deriving a well-defined $\mathit{defines}_p$ function from the above, we must take the greatest rather than the least fixpoint of the transformed function, because we wish to find *definite* definition. This can be illustrated by the following example:

```
setlast([Last]) ::-
        $last := Last.
setlast([E1,E2|L]) ::-
        setlast([E2|L]).
```

If we were to compute the least fixpoint of the transformed $\mathit{defines}_p$ by Kleene iteration, we would find $\mathit{lfp}(\mathit{defines}_p)(\texttt{setlast}) = \emptyset$, which is clearly not correct, as any successful execution of a call to `setlast` must assign the global variable `$last`. However, $\mathit{gfp}(\mathit{defines}_p)(\texttt{setlast}) = \{\mathit{last}\}$, as we expect.

Now we are equipped to find the sets we are looking for: for each predicate, the set of global variables that it may need to take as input, and the set that it may be required to provide as output. These are defined by the functions $\mathit{provides}_p : \mathit{preds} \to 2^{\mathit{globs}}$, $\mathit{provides}_c : \mathit{conjs} \to 2^{\mathit{globs}}$, $\mathit{needs}_p :\to 2^{\mathit{globs}}$, and $\mathit{needs}_c : \mathit{conjs} \to 2^{\mathit{globs}}$ as follows:

$$
\begin{aligned}
\mathit{provides}_p(p) &= \bigcup_{H::-B} \left(\begin{array}{c} \mathit{changes}_p(p) \cap ((\mathit{provides}_p(q) \setminus \mathit{defines}_g(B')) \\ \cup\, \mathit{needs}_c(B') \end{array} \right) \\
& \;\textbf{where } B = G_1, \ldots, G_n \textbf{ and } B' = G_{i+1}, \ldots, G_n \\
& \;\;\textbf{and } 1 \le i \le n \textbf{ and } \mathit{pred}(G_i) = p \textbf{ and } \mathit{pred}(H) = q \\
\mathit{provides}_c(B) &= \bigcup_{1 \le i \le n} \mathit{provides}_p(\mathit{pred}(G_i)) \textbf{ where } B = G_1, \ldots, G_n
\end{aligned}
$$

$$
\begin{aligned}
\mathit{needs}_p(p) &= (\mathit{uses}_p(p) \cup \mathit{provides}_p(p)) \setminus \mathit{defines}_p(p) \\
\mathit{needs}_c(G) &= (\mathit{uses}_c(G) \cup \mathit{provides}_c(G)) \setminus \mathit{defines}_c(G)
\end{aligned}
$$

The definition of $\mathit{provides}_p$ given here may benefit from an illustrative example. Consider the clause:

```
p ::- q, r, s.
```

Of the global variables `q` changes, `q` must provide any variables that `(r,s)` needs, plus any that `p` provides and `(r,s)` does not define. That means that of the global variables `q` changes, it must provide all the variables that `r` uses, plus the variables `s` uses that `r` does not create, plus what `p` provides that neither `r` nor `s` create.

We need to define one more mapping for error-checking purposes. It is possible that predicate p needs a global variable v, but v may not have been assigned before some of the calls to p. This is, of course, an uninitialized variable error. In order to detect such errors, we define the set of variables that are definitely available at all calls to a predicate p as:

$$\begin{array}{rcl} available(p) & = & \displaystyle\bigcap_{H::-G_1,\ldots,G_n} \bigcap_{1\leq i\leq n} \left(\begin{array}{c} available(pred(H)) \\ \cup\, defines_c(G_1,\ldots,G_{i-1}) \end{array} \right) \\ & & \textbf{where } pred(G_i) = p \end{array}$$

As with $defines_p$, we must define *available* as the *greatest* fixpoint of the transformed version of the above definition.

At last we can define the sets of global variables that must be input to and output from each predicate, and the set of global variables that *should* be input, but are not available (the error set). These are defined by the mappings $inputs : preds \rightarrow 2^{globs}$, $outputs : preds \rightarrow 2^{globs}$ and $errors : preds \rightarrow 2^{globs}$ as follows:

$$\begin{array}{rcl} inputs(p) & = & needs(p) \cap available(p) \\ outputs(p) & = & provides(p) \\ errors(p) & = & needs(p) \setminus available(p) \end{array}$$

4.3 Our Implementation

One issue that arises when implementing global variables for use in the real world is how they should interact with a module system. We believe that truly global variables, which can be set and accessed from multiple modules, are probably a bad idea. At the very least, they should come under the control of the module system, which would require a new syntax for exporting and importing global variables. Rather than do this, we have chosen to force the use of a global variable from outside the unique module which defines it to go through exported predicates. Global variables are only global to a single module.

The analysis described in section 4.2 is clearly a global analysis, which means that we must have access to the entire program before we can generate any code. In order to be convenient for the Prolog programmer to use, the translator is implemented using Prolog's `term_expansion` facility. This is a common, though nonstandard, Prolog extension that allows users to define their own translations from the clauses and terms appearing in a source file into the actual clauses of the program. In order to effect a global analysis, we make use of two specific features of Quintus Prolog: the ability of `term_expansion` to translate a single term into a list of clauses, and the fact that Quintus Prolog actually invokes `term_expansion` on the `end_of_file` marker. The former is a fairly common extension to `term_expansion`, and the latter is a small Quintus-specific enhancement[1]. We use these features by translating each clause in the file into the empty list of clauses, while saving away the clause; at the end of the file, we perform the analysis, translate the program, and emit the translation of the entire file. Note that the translator could certainly be implemented as a source file to source file translator, though this would be considerably less comfortable for users.

This approach has the important consequence that the analysis is not truly global; only a single file is analyzed at a time, which limits some uses of global variables, even for variables only used in a single file. For example, if a predicate in module `A` defines a global variable and then calls a predicate in module `B`, which in turn calls another predicate in module `A`, the analysis of module `B` will be unable to determine how `A`'s global variable is passed through `B`. Also, if a call from module `A` to module `B` which defines a global variable is followed by another call from `A` to `B`, the analysis of `A` will be unable to determine how `B`'s variable is passed through `A`. We discuss fixing these shortcomings in section 5. In the mean time, interfaces between files must be free of global variables; all information passed between files must be passed through arguments.

It is our belief that forcing programmers to analyze and translate their entire program at once would be a sufficient inconvenience that most would not bother to use such a tool. It is possible to write an incremental truly global analyzer and translator, but it would be more difficult, and would preclude the `term_expansion` approach. This would be an interesting avenue for future investigation.

We also make another restriction: we require that the global analysis *prove* that each global variable will be assigned before it is accessed. If this cannot be proved, an error is reported. If it is possible for a predicate to refer to a variable before it is assigned a value, we conservatively assume that it does, and so a value is required to have been assigned before that predicate is called. If a predicate may or may not assign a value to a variable, we conservatively assume that it does not, and so a call to such a predicate will not suffice to guarantee that that variable has been assigned. Since our translation is based on analysis, and since the reachability of goals in a clause is in general undecidable, this is the best we can do. We believe that this restriction will prove to be a positive one, since it means the analysis can always report uninitialized variable problems at compile time.

The fixpoint definitions of the functions of section 4.2 provide an al-

[1] A Prolog system that already supports `term_expansion` can usually be extended fairly easily to support these two features. Many Prolog vendors would probably make these extensions if their customers insisted.

gorithm for computing the desired *inputs*, *outputs*, and *errors* sets. But the obvious algorithm can be significantly improved, to provide a reasonably efficient translator. We compute the strongly-connected components (SCCs) of the program call graph, and perform a topological sort of the SCCs. This allows us to compute the $changes_p, uses_p$ and $defines_p$ sets bottom up, one SCC at a time. Because of nature of the definitions of the $changes_p$ and $uses_p$ sets, all of the predicates in a given SCC will have as their $changes_p$ and $uses_p$ sets the unions of the $changes_p$ and $uses_p$ sets of the predicates they call, plus the unions of the *appears* and *assigns* sets of the goals appearing in the clauses, respectively. Thus we may find the fixpoint without Kleene iteration.

The $defines_p$ set cannot be computed as easily, but in the very common special case of SCCs with only one predicate, a single Kleene iteration will always find the greatest fixpoint for an SCC, so we can avoid the second iteration to confirm the fixpoint.

Determining the $needs_p$ and $provides_p$ sets is harder. Firstly we notice that we can compute the first approximation for $needs_p$ bottom-up while computing $changes_p$, $uses_p$ and $defines_p$, by ignoring $provides_p$ (as it will be empty on the first Kleene iteration). Once the full bottom-up pass is completed, we make a top-down pass computing $provides_p$ and $needs_p$. This is done by processing the clauses in an SCC from right to left, keeping track as we go of the set of global variables that must be provided by the goal we are considering or an earlier goal in the clause. We call this set n. For the rightmost goal in a clause, n is just (the current approximation of) the $provides_p$ set of the predicate whose clause is currently being analyzed. As we consider a call to predicate p, we know that p must provide at least $changes_p(p) \cap n$, so we add this to $provides_p(p)$. If this adds any new global variables to $provides_p(p)$, then we also recompute $needs_p(p)$ to ensure we always use the best approximation available. The set n for the next goal to the left will be $n \setminus changes_p(p) \cup needs_p(p)$.

Unfortunately a single pass of this will not necessarily be enough. In fact, $provides_p(p)$ depends not only on $provides_p(q)$ for all predicates q that call p, but also on $needs_p(r)$ for all predicates r that are called to the right of a call to p. This means that a fixpoint iteration over an individual SCC will not even work.

Therefore we adopt a dependency-based strategy for the top-down pass. We define each predicate's *dangerzone* as $changes_p \setminus provides_p$. These are the variables that a predicate is not currently expected provide, but might have to. A predicate p's *dangermap* is a map from variables to sets of predicates, mapping a variable v that p changes but neither defines nor needs to the set of all the predicates q which are called to the left of some call to p in the program, where q is the predicate called nearest to the left of p which has v in its *dangerzone*. When we find that a predicate needs a variable we did not know it needed, we look up that variable in the predicate's *dangermap* and add that variable to the $provides_p$ sets of each predicate in the corresponding set. Then for each of them we refigure its $needs_p$ set. If that changes, we apply this procedure recursively. Termination is guaranteed because at each step we add at least one variable to some predicate's $provides_p$ set, and there is a finite set of global variables that can be in any predicate's $provides_p$ set. We compute the *dangermap* as part of the top-down pass, adding to it whenever we encounter a call that does not provide a global variable that it changes. In this way, a single top-down pass will compute the least fixpoint of $needs_p$ and $provides_p$.

The *available* set is also computed in the same top down pass, but it is computed left to right. Again, a single iteration is sufficient for single-predicate SCCs, but more iterations may be needed for larger SCCs. At the same time, we compute the dangers in the clause from left to right, so that as we process goals from right to left we can update each called predicate's *dangermap* to reflect the dangers of that clause.

5 Conclusions and Future Work

We have presented a proposal and a formal semantics for logic programming with global variables. The addition of global variables to logic programming allows real world programs to be expressed more concisely, avoiding some of the problems associated with having predicates with many arguments. Although conjunction in this logic is neither commutative nor absorptive, the practical effect of this is rather small as long as the user has easy access to a list of the global variables changed and used by each predicate. By using a preprocessor to generate from a program with global variables essentially the program that would have been written without them, we show that there is no loss of performance attributable to global variables.

There are a few extensions and improvements to this language and its implementation that we may consider adding in the future. Firstly, as mentioned earlier, it would be desirable to allow global variables to exist over cross-file calls. This would require no change to the semantics — in fact it would make the implementation better reflect the semantics — but would be impossible to implement generally while maintaining file-at-a-time compilation. We could support this by allowing users to declare which global variables should be allowed to survive through cross-module calls (or perhaps this should be the default if the scoping extension mentioned below is implemented). Because our implementation uses both bottom-up and top-down analysis, we cannot implement our full analysis for cross-module calls, but we can compute $uses_p$, $changes_p$, and $defines_p$ if we always analyze a module before analyzing the modules that use it. (Of course, this cannot be done for interdependent modules, so another solution would be needed in this case.) We can then take $changes_p$ as a safe approximation for $provides_p$ for exported predicates.

If cross-file global variables are to be supported, it would also sometimes be useful to be able to declare an initial value for a global variable. This would allow a global variable to maintain some internal state for a module without any client modules needing to know about it.

It would sometimes be convenient to have a special syntax to assign a new value to a global variable during unification. Consider writing code to increment an integer stored in a global variable `$i`: `I1 is $i + 1, $i := I1`. This would be much more convenient if one could pass an indication that the result of the call to `is`/2 should be stored directly into `$i`. A term `$:=i` would specify a term which would unify with any term and have the effect of binding `$i` to that term; for example `$:=i is $i + 1` would increment `$i` in a single goal. It should probably be considered a semantic error for multiple `$:=` terms naming the same variable to appear in a single goal. Other than this caveat, this extension should not present any significant difficulty for the semantics, nor would it be difficult to implement.

The programmer may sometimes want for reasons of robustness to specify that a global variable should not leave a certain scope. This could prevent an assignment of a global variable from accidentally affecting a part of the computation it is not meant to participate in. Again, it would not be difficult to extend the semantics to handle this, nor would it be difficult to implement.

We expect to make the Prolog version of the translator publicly available soon. It will be published on the worldwide web accessible from the URL: `http://www.cs.mu.oz.au/~pets/`.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] John Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[3] Peter Van Roy. A useful extension to Prolog's definite clause grammar notation. *ACM SIGPLAN Notices*, 24(11):132–134, November 1989. URL: `http://www.info.ucl.ac.be/people/PVR/edcg_article.ps`.

[4] Paul Tarau. *BinProlog 5.25 User Guide*. Departement d'Informatique, Université de Moncton, Moncton, Canada. Part of the BinProlog 5.25 distribution, available from `ftp://clement.info.umoncton.ca/BinProlog/`.

[5] Paul Tarau, Veronica Dahl, and Andrew Fall. Backtrackable state with linear affine implication and Assumption Grammars. In *LNCS, Proceedings of Asian Computing Science Conference*, Singapore, December 1996. Springer.