LOGIK

Grigore Roşu and Traian Florin Şerbănuţă (`{grosu,tserban2}@illinois.edu`)
University of Illinois at Urbana-Champaign

Abstract

This is the $\mathbb{K}$ semantic definition of LOGIK, a trivial language capturing the essence of the logic programming paradigm. In this definition, we explicitly focus on simplicity and mathematical clarity, not on advanced logic programming features or performance. Those are covered in the LOGIK++ extension under `examples/logik++`.

Specifically, a LOGIK program consists of a sequence of *Horn clauses* of the form

```
P :- P1, P2, ..., Pn .
```

followed by a *query* of the form

```
?- Q1, Q2, ..., Qm .
```

where `P`, `P1`, `P2`, ..., `Pn`, `Q1`, `Q2`, ..., `Qm` are *literals*. The symbol "`:-`" is read "if". A literal has the form `p(T1,T2,...,Tk)`, where `p` is a *predicate symbol* and where `T1,T2,...,Tk` are *terms*. Terms are built as usual, with *operation symbols* and *variables*. A common convention in logic programming languages, also adopted here, is that variables are capitalized and operation symbols are not. Operations with zero arguments are called *constants* and are written without parentheses, that is, `c` instead of `c()`. Horn clauses without conditions, called *facts*, are written without "`:-`", that is, "`P.`" instead of "`P :- .`".

For example, the LOGIK program below gives a few facts about a `parent` predicate, then several clauses defining some useful predicates including an `ancestor` predicate, and finally a query asking for those who both have ancestors and are ancestors themselves in the `parent` relation:

```
parent(david,john).
parent(jim,david).
parent(steve,jim).
parent(nathan,steve).

grandparent(A,B):-
  parent(A,X),
  parent(X,B).

ancestor(A,B):-
  parent(A,X),
  parents(X,B).

parents(X,X).
parents(A,B):-
  ancestor(A,B).

both(X) :- ancestor(A,X), ancestor(X,B).

?- both(X).
```

Above, we only have constant operation symbols, so these and variables are the only terms that can be used in predicates. As expected, the LOGIK program above will give us three solutions for `X`: `david`, `steve`, and `jim`. If we inline the `both(X)` predicate in the query, that is, if we replace the query with "`?- ancestor(A,X), ancestor(X,B).`" then we get 10 solutions, one for for each triple `A`, `X`, and `B` satisfying both predicates `ancestor(A,X)` and `ancestor(X,B)`.

As another example, the program below defines an `append` predicate followed by a simple goal:

```
append(nil,L,L).
append(cons(H,T),L,cons(H,Z)) :- append(T,L,Z).

?- append(cons(a,nil), cons(b,nil), V).
```

Besides the predicate symbol `append`, the program above also includes a constant symbol `nil` and a binary operation symbol `cons`. Additionally, the query also includes two more constants, `a` and `b`. The capitalized identifiers are all variables. As expected, the LOGIK program above yields only one solution, namely `V = cons(a,cons(b,nil))`. On the other hand, if we change the query to:

```
?- append(L1, cons(a,L2), cons(a,cons(b,cons(a,nil)))).
```

then LOGIK yields two solutions: one where `L1` is `cons(a,cons(b,nil))` and `L2` is `nil`, and another where `L1` is `nil` and `L2` is `cons(a,cons(b,nil))`.

The programs above all generated *ground solutions*, that is, solutions where the query variables are mapped to ground terms (i.e., terms without variables). Let us now consider the following query:

```
?- append(cons(a,nil), Y, Z).
```

There are obviously infinitely many ground solutions for the query above, e.g., `Y = nil` and `Z = cons(a,nil)`, `Y = cons(a,nil)` and `Z = cons(a,cons(a,nil))`, `Y = cons(b,nil)` and `Z = cons(a,cons(b,nil))`, `Y = cons(c,cons(b,nil))` and `Z = cons(a,cons(c,cons(b,nil)))`, etc. However, all the ground solutions for the query above can be elegantly characterized by the property that `Z` is bound to a list starting with `a` and followed by the list that `Y` is bound to. This property can in fact be described as a *symbolic solution* to the query: `Z = cons(a,Y)` or, equivalently, `Y = Symb` and `Z = cons(a,Symb)`. It is possible to define a "more general than" relation on such symbolic solutions, in the sense that the more particular solution can be obtained as a specialization/substitution of the more general one, and then it can be shown that the above is the *most general solution* to the stated query. Logic programming languages, including our LOGIK, attempt to always compute such most general solutions.

Logic programming languages are highly non-deterministic, in that several Horn clauses may be used at the same time, each possibly resulting in a different solution. Implementations of logic programming languages consist of complex, optimized search and indexing algorithms, which we are not concerned with here. Instead, we here take advantage of $\mathbb{K}$'s builtin support for search. Specifically, to find all the solutions of a LOGIK program, we have to use `krun` with the option `-search`. However, note that some programs have infinitely many solutions which cannot relate to each other by the "more general" relation. For example, the query

```
?- append(L1, cons(a,L2), L3) .
```

To address such cases and terminate, logic programming languages allow the user to choose how many solutions to be computed and displayed. In LOGIK, we can use the `-bound` option of `krun` for this purpose.

Finally, note that some queries have no solution. In some cases that is easy to detect by exhaustive analysis, such as for the following query:

```
?- append(cons(a,L1), L2, cons(b,L3)).
```

Logic programming languages, including LOGIK, terminate in such cases and report a no solution answer. However, there are cases where exhaustive analysis is not sufficient, such as for the query:

```
?- append(cons(a,L), nil, L).
```

In such cases, logic programming languages do not terminate. While one may devise techniques to detect non-termination in some cases, one cannot do it in general (same like for all Turing-complete languages).

MODULE LOGIK

Unification is at the core of logic programming. Here we are going to use the predefined unification procedure (the same one we used in the type inferencers in Tutorial 5.

Syntax

The syntax of LOGIK is straightforward: a program is a sequence of Horn clauses followed by a query:

SYNTAX *Term* ::= *Variable*
| *Literal* [klabel('LiteralAsTerm)]
| *Literal*(*Terms*)

SYNTAX *Terms* ::= $List\{$*Term*, ","$\}$

SYNTAX *Clause* ::= *Term* `:-` *Terms* `.`
| *Term* `.`

SYNTAX *Query* ::= `?-` *Terms* `.`

SYNTAX *Pgm* ::= *Query*
| *Clause* *Pgm*

Variables and literals are defined as tokens following the conventions used in Prolog (variables start with _ or capital letter, while literals start with lower case letters):

SYNTAX *Variable* ::= $Token\{[\backslash\_A-Z][a-zA-Z0-9\backslash\_]*\}$ [onlyLabel]

SYNTAX *Literal* ::= $Token\{[a-z][a-zA-Z0-9\backslash\_]*\}$ [onlyLabel]

Normalizing the definition

The following simple macros ensure that all clauses will have both a conclusion and a condition, and that all predicate and operation symbols take a (potentially empty) list of arguments. These will simply the semantics later on, by not having to treat special cases.

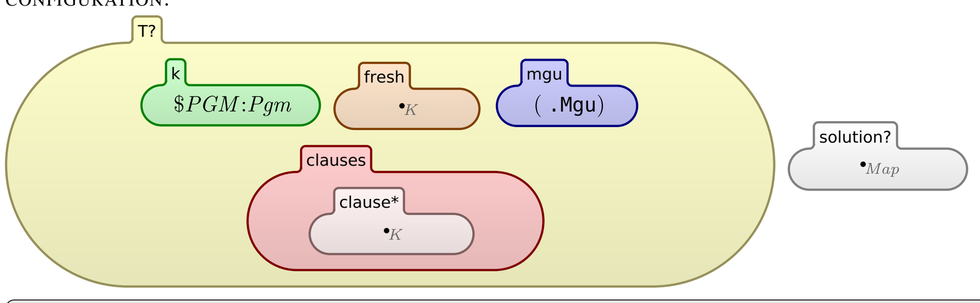
RULE $\dfrac{L\ \texttt{.}}{L\ \texttt{:-}\ \bullet_{Terms}\ \texttt{.}}$ [macro]

RULE $\dfrac{{}'LiteralAsTerm(X{:}Literal)}{X(\bullet_{Terms})}$ [macro]

Configuration

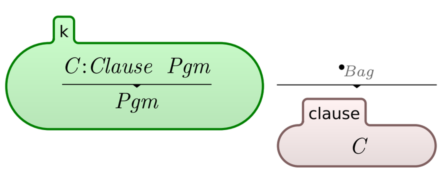
The configuration stores each clause in its own cell for easy access, and the most general unifier in a cell named `mgu`, same like the type inferencers. The `k` cell holds the query and the `fresh` cell holds a fresh clause instance to be attempted on the next query item. To more easily read the solutions, we add a second top-level cell, `solution`. Both top cells are optional. Indeed, we start with the main top cell and, when a solution is found, we move it into the `solution` cell and discard the main cell.

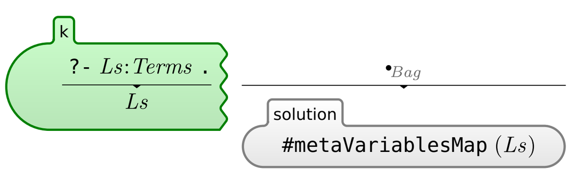
CONFIGURATION:



Pre- and post-processing

Before we launch the semantics, we first scan the given program and place each clause in its own cell, and then place the query in the k cell, creating the solution cell, too:

RULE 

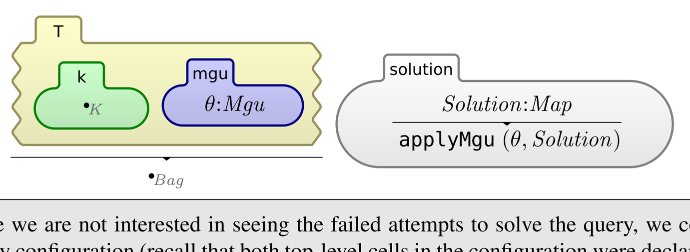
RULE 

We also sequentialize the goals for easier processing:

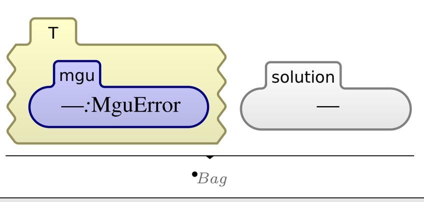
RULE $\dfrac{L{:}Term\ \texttt{,}\ Ls{:}Terms}{L \curvearrowright Ls}$

RULE $\dfrac{\bullet_{Terms}}{\bullet_{K}}$

When all the goals are solved, indicated by the empty `k` cell, the calculated most general unifier (mgu) is in the `mgu` cell. In that case, to ease reading of the final solution we apply the mgu to the solution map and delete the rest of the configuration:

RULE 

Since we are not interested in seeing the failed attempts to solve the query, we collapse all the error configurations into an empty configuration (recall that both top-level cells in the configuration were declared optional). This way, if we see an empty configuration when we search for all solutions, we know that some attempts failed (but we do not know which ones).

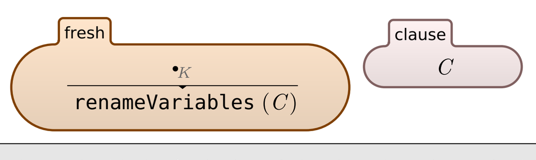
RULE 

Semantics

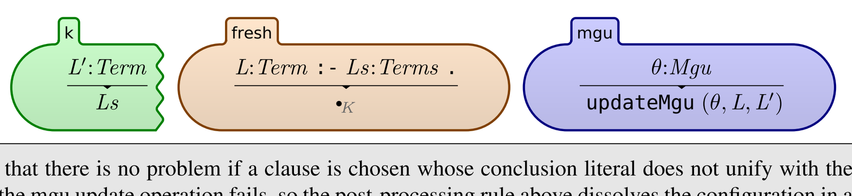
Once all the infrastructure is in place, the actual semantics of LOGIK is quite simple. All we have to do is to pick some (fresh instance of a) clause, then unify its conclusion with the first query literal, and then replace that literal with condition of the clause. The intuition here is the following: to satisfy the first literal in the query, we need to find some instance of some clause that matches it, and then to similarly show that we can satisfy the conditions of that clause. Mathematically, this is an instance of the proof principle called *resolution*: if $p \vee q$ and $\neg p \vee r$ hold, then so does $q \vee r$. We let it as an exercise to the reader to see how the two relate (hint: assume the negation of the goal together with all the clauses, and then derive *false*).

The following two rules are tightly connected and they together perform the following core task: pick a fresh instance of a clause which unifies with the first goal item, then add its conditions as new goals. By a "fresh instance" of a clause we mean one whose variables are renamed with fresh names; we need that in order to avoid undesired unification conflicts due to particular names chosen for variables in the original program, as well as conflicts due to subsequent uses of the same clause. It is safe to rename the variables in a clause, because clauses are universally quantified in their variables. This process of creating a fresh instance of a clause is similar to how we created fresh instances of type schemas in the higher-order type inferencer discussed in Tutorial 5. Indeed, we can safely regard clauses as "clause schemas" comprising infinitely many instances, one for each context.

Pick a clause and generate a fresh instance of it when the `fresh` cell is empty:

RULE  [transition]

Unify the goal with the fresh clause's head, replace the goal with the clause body, and empty the `fresh` cell (so that another clause can be chosen using the rule above):

RULE 

Note that there is no problem if a clause is chosen whose conclusion literal does not unify with the first goal literal. In this case the mgu update operation fails, so the post-processing rule above dissolves the configuration in an empty one. The search option of `krun` will systematically try all clauses, so no solution is missed. Of course, the above is not the most efficient way to implement a logic programming language, but recall that our objective here was to present a simple and mathematically clean solution. We encourage the interested reader to consult the LOGIK++ language definition for a more efficient definition of a richer logic programming language.

END MODULE