Department of Computer Science, University of Otago

Technical Report OUCS-2017-06

Logic Programming Modules as Possible Worlds

Author:

Richard O'Keefe
Department of Computer Science, University of Otago, New Zealand

---

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.php

Logic Programming Modules as Possible Worlds

Richard A. O'Keefe
Computer Science, Otāgo

June 2017

Abstract

Existing module systems for logic programming, such as ISO Prolog part 2 or Mercury are basically syntactic in nature. They are intended to be compatible with the usual semantics of logic programming. Taking a semantic view leads to the idea of modules as possible worlds, and cross-module calling as a modal operator. This opens up additional possibilities, including some useful for software engineering.

1 Modules for pure logic programs

A (pure) logic program is a set of positive Horn clauses. Predicates are identified by their name and arity. For example,

reverse(L, R) ← reverse(L, nil, R)

reverse(nil, R, R) ←
reverse(cons(H,T), R0, R) ← reverse(T, cons(H,R0), R)

contains two predicates, reverse/2 and reverse/3.

Pure logic programs contain no higher-order predicates, so a module system can be very simple. A module has a name, a set of exports, and a set of positive Horn clauses. A call to a predicate defined in another module is distinguished by a module prefix. For example,

← module(lists)
← export(reverse/2)
. . . as before . . .

← module(demo)
← export(palindrome/1)

palindrome(L) ← lists:reverse(L, L)

This in effect renames every predicate f/n defined in module m to $m : f/n$. In the example, it is as if we had

lists:reverse(L, R) ← lists:reverse(L, nil, R)

lists:reverse(nil, R, R) ←
lists:reverse(cons(H,T), R0, R) ← lists:reverse(T, cons(H,R0), R)

demo:palindrome(L) ← lists:reverse(L, L)

This can of course be handled by making the module prefix an additional argument. See section 5 for details.

reverse(lists, L, R) ← reverse(lists, L, nil, R)

reverse(lists, nil, R, R) ←
reverse(lists, cons(H,T), R0, R) ← reverse(lists, T, cons(H,R0), R)

palindrome(demo, L) ← reverse(lists, L, L)

a translation which allows a module prefix to be a variable.

The rôle of the set of exports is to provide encapsulation by allowing cross-module calls only to predicates which the author intended to make available to other modules. Prolog and Mercury allow import control as well as export control. But all of that only affects which cross-module calls are allowed, not what they mean.

Since control over which cross-modules can be separated from the semantics of those calls, we can concentrate on the latter.

Higher-order operations can to some extent be handled by adding, for every predicate $m : f/n$, a clause

call(m, $f(X_1, \ldots, X_n)$) ← $m : f(X_1, \ldots, X_n)$

2 Problems

Prolog allows run-time changes to the program. A cross-module call might be forbidden at the time it is compiled but allowed at the time it is executed, or *vice versa*.

Higher-order predicates make life much harder. A type system such as the one in Mercury can solve this problem, or the `meta_predicate` declaration introduced by the author at Quintus, which is basically a "poor man's type declaration".

A problem which Prolog shares with Mercury, Erlang, and Lisp is that if a module exports any predicate, *every* module can call it.

3 Modal Logic

Classical modal logic extends propositional calculus (or first order predicate calculus) with *modal operators*, notably the "possibly" operator $\Diamond$ and the "necessarily" operator $\Box$.

Truth in modal logic is relative to a *world*. Worlds are part of a *frame* (G, R), which consists of a set G of worlds and an *accessibility relation* R, where hRt means that world t is accessible from world h.

Restricting ourselves to propositional modal logic, we have

- $w \models p$ if proposition p holds in w
- $w \models \neg p$ iff $w \not\models p$
- $w \models p \wedge q$ iff $w \models p$ and $w \models q$
- $w \models p \vee q$ iff $w \models p$ or $w \models q$
- $w \models p \Rightarrow q$ iff $w \models q \vee \neg p$
- $w \models \Box p$ iff $\forall u \in G(wRu \Rightarrow u \models p)$
- $w \models \Diamond p$ iff $\exists u \in G(wRu \wedge u \models p)$

If this is all we have, we get a logical system called K. By placing restrictions on R, we get a range of systems allowing more theorems than K.

4 Application to logic programming

The key steps are

- world = module
- worlds can be recognised by their names
- cross-module call = $\Diamond$
- R provides more fine-grained access control
- `module` declarations provide G
- `accessible_to` declarations provide R.

Different facts may be true in different worlds. This is exactly what we want for modules.

Modules are named. Let us rule that a module must contain one and only one clause of the form

module$(m) \leftarrow$

and that at its beginning. This will be generalised shortly.

We need also to express the accessibility relation. Every module, say m, must contain at least one clause of the form

accessible_to(c) $\leftarrow$

meaning that cRm is true. This too will be generalised shortly.

Now we can express the semantics of $m : g$ in module c as

$$\Diamond(\text{module}(m) \wedge g)$$

where the accessibility relation is defined by the accessible_to/1 clauses.

5 Translation to plain Horn clauses

The module(m) clauses are unmodified.

An accessible_to(c) clauses is translated to accessible_to(m, c).

A clause $h(T_1, \ldots, T_n) \leftarrow B$ is translated to $h(m, T_1, \ldots, T_n) \leftarrow B[\![m]\!]$.

$$\begin{aligned}
T[\![(a, b)]\!]m &= T[\![a]\!]m, T[\![b]\!]m \\
T[\![(a; b)]\!]m &= T[\![a]\!]m; T[\![b]\!]m \\
T[\![(a \rightarrow b)]\!]m &= T[\![a]\!]m \rightarrow T[\![b]\!]m \\
T[\![m : g]\!]m &= T[\![g]\!]m \\
T[\![m' : g]\!]m &= \text{accessible_to}(m, m'), T[\![g]\!]m' \\
T[\![p(T_1, \ldots, T_n)]\!]m &= p(T_1, \ldots, T_n) \text{ if } p/n \text{ is built-in} \\
T[\![p(T_1, \ldots, T_n)]\!]m &= p(m, T_1, \ldots, T_n) \text{ otherwise}
\end{aligned}$$

Our running example now becomes

```
module(lists)

reverse(lists, L, R) ← reverse(lists, L, nil, R)

reverse(lists, nil, R, R) ←
reverse(lists, cons(H,T), R0, R) ← reverse(lists, T, cons(H,R0), R)

module(demo)

palindrome(demo, L) ←
      accessible_to(lists, demo),
      reverse(lists, L, L)
```

which of course will not work, because demo is not accessible to lists. We need a declaration such as

accessible_to(demo). % or
accessible_to(_).

in the lists module.

6 Generalisation

Nothing in the translation above depends on module names being atoms, or even ground. Allowing module names to be compound terms can give us some of the power of ML modules. Here is a simple sorting example.

module(basic)
accessible_to(_)

cmp(O, X, Y) $\leftarrow$ compare(O, X, Y).

module(second(M))
accessible_to(_)

cmp(O, pair(_,X), pair(_,Y)) $\leftarrow$
    M:cmp(O, X, Y)

module(sort(L))
accessible_to(user)

insert(X, nil, cons(X,nil)) $\leftarrow$
insert(X, cons(H, T), S) $\leftarrow$
    L:cmp(O, X, H),
    insert(O, X, H, T, S)

insert('<', X, H, T, cons(X,cons(H, T))) $\leftarrow$
insert('=', X, H, T, cons(X,cons(H, T))) $\leftarrow$
insert('>', X, H, T, cons(H, S)) $\leftarrow$
    insert(X, T, S)

isort(R, S) $\leftarrow$ isort(R, nil, S)
isort(nil, S, S) $\leftarrow$
isort(cons(X, R), S_0, S) $\leftarrow$
    insert(X, S_0, S_1),
    isort(R, S_1, S) module(user)

$\leftarrow$ sort(basic):isort(cons(3,cons(1,cons(4,nil))), Ans)

$\leftarrow$ sort(second(basic)):isort(
    cons(pair(a,3),cons(pair(b,1),cons(pair(c,4),nil))), Ans)

Normally something like this would be done in Prolog using meta-call, *e.g.*,

```
:- meta_predicate
    isort(3, +, -),
        isort(+, +, -, 3),
            insert(+, +, -, 3),
```

```
                insert(+, +, +, +, -, 3),
    second(3, -, +, +).

isort(C, R, S) :-
    isort(R, [], S, C).

isort([], S, S, _).
isort([X|R], S0, S, C) :-
    insert(X, S0, S1, C),
    isort(R, S1, S, C).

insert(X, [], [X], _).
insert(X, [H|T], S, C) :-
    call(C, O, X, H), % meta-call
    insert(O, X, H, T, S, C).

insert(<, X, H, T, [X,H|T], _).
insert(=, X, H, T, [X,H|T], _).
insert(>, X, H, T, [H|S],   C) :-
    insert(X, T, S, C).

basic(O, X, Y) :- compare(O, X, Y).
second(M, O, pair(_-X, _-Y)) :- call(M, O, X, Y).

?- isort(basic, [3,1,4], Ans).
?- isort(second(basic), [a-3,b-1,c-4], Ans).
```

This passes around a term C representing part of a call to a predicate. (In this case, the last three arguments are missing.) The module-as-world approach instead passes around a term M naming a module which can export any desired number of predicates.

7 Conclusion

Thinking of modules as possible worlds led fairly directly to a simple scheme that was originally interesting because of the potential for handling "versions and variations" without the need for any preprocessor, which will be addressed in a later note. The fact that it offers some of the power of ML structures and functors was a pleasant bonus.