A Prolog Interpreter for F-Logic

Michael Lawley

School of Computing and Information Technology
Griffith University
Nathan, Queensland, Australia 4111
lawley@cit.gu.edu.au

March 29, 1994

1 Introduction

The Frame Logic or F-logic of Kifer, Lausen and Wu [3] is currently the most developed and complete formal model for deductive object-oriented languages. However, it has a complex semantics and the meaning of apparently simple programs is often difficult to ascertain. Furthermore, the very comprehensiveness of F-logic leaves doubts as to whether the language is implementable. Indeed, Kifer et al. suggest various options for implementing a programming language based on a subset of F-logic.

Implementing a substantial subset of F-logic is therefore an important task. It would clarify the language issues involved in combining the object-oriented paradigm with logic programming and deductive databases and shed light on the implementation issues with regard to tractability and efficiency, and the properties of the language, namely the interaction of overriding and deduction.

In this paper we describe our experience in defining such an interpreter for F-logic. Out interpreter implements inheritance, overriding, functional and set-valued methods, types, negation and relations. We have used the implementation for small examples and to implement a query translator for heterogeneous databases described by F-logic programs [4].

In section 2 we present a brief, informal, overview of F-logic. In section 3 we give an overview of the structure of the interpreter. Then, in section 4, we introduce a minimal interpreter which implements inheritance and deduction but none of the interesting features of F-logic. This is followed, in section 5, by a series of modifications to the basic interpreter, each implementing an additional feature of F-logic. At each step we highlight aspects of F-logic as revealed by the implementation. Finally, we draw together the different threads to indicate what we have learnt about F-logic and its implementation requirements, and what this suggests for the design of both an efficient and expressive declarative object-oriented language. The complete version of the interpreter is given in the appendix.

2 Overview of F-logic

In this section we give a brief overview of the syntax and semantics of F-logic. Obviously we cannot cover F-logic in all its details. The reader is referred to [3] for these.

An F-logic term may be one of several kinds: an *id-term*, an *f-term*, or a *p-term*. In turn, an f-term is either an *isa-term*, a *data-term*, or a *signature-term*.

An id-term $jane$ or $dept(cs)$ denotes a class or object. F-logic does not distinguish between classes and objects.

The isa-term $o : cl$ states that the object or class o is an instance or subclass of the class cl.

The data-term $o[m@arg_1, arg_2 \rightarrow v]$ states that the method m, when applied to object o with arguments arg_1, arg_2, has value v. The single-headed arrow indicates that m is a functional or single-valued method. The data-term $o[m' \twoheadrightarrow \{v1, v2\}]$ states that the method m', when applied to object o with no arguments, is a set containing $v1$ and $v2$. In this case the double-headed arrow indicates that m' is a set-valued method. Note that m' could inherit other set elements from the superclasses of o.

The signature-term $o[m@t_1, t_2 \Rightarrow \{t_3, t_4\}]$ states that the method m, applied to object o takes arguments of type t_1 and t_2 and returns a value which is of type t_3 and t_4. A double headed arrow $\Rightarrow\!\!>$ would indicate a set-valued method.

The p-term $p(a_1, a_2)$ is used in the same way as predicates in Prolog. It is used to integrate relations cleanly with the rest of F-logic.

Note, variables may appear anywhere an id-term may appear, including as a method name or argument.

An F-logic program consists of a set of rules or clauses. A rule has a *head*, which is an f-term or p-term, and a *body*, which is a conjunction of, possibly negated, f-terms and p-terms. These rules behave similarly to Prolog rules: if the body can be proved true then the head can be inferred.

The following is an example of an F-logic program.

$$employee : person. \tag{1}$$
$$jane : employee. \tag{2}$$
$$employee[salary@year \Rightarrow integer]. \tag{3}$$
$$person[children \Rightarrow\!\!> person]. \tag{4}$$
$$person[parent \Rightarrow\!\!> person]. \tag{5}$$
$$jane[salary@1990 \rightarrow 34000]. \tag{6}$$
$$jake[children \twoheadrightarrow \{john, lisa\}]. \tag{7}$$
$$C[parent \twoheadrightarrow \{P\}] \leftarrow P[children \twoheadrightarrow \{C\}]. \tag{8}$$

Various complex nestings of data, signature, and isa terms are allowed, but these are merely syntactic sugar, adding no expressive power to the language. These are described as follows.

- The complex data term $o[m1 \rightarrow v1 \ ; \ m2 \rightarrow v2]$ is equivalent to the conjunction $o[m1 \rightarrow v1]$ & $o[m2 \rightarrow v2]$.

- The complex data term $o1[m1 \rightarrow o2[m2 \rightarrow o3]]$ is equivalent to the conjunction $o1[m1 \rightarrow o2]$ & $o2[m2 \rightarrow o3]$.
- The data term $o[m \twoheadrightarrow \{a_1, \ldots, a_k\}]$ is equivalent to the conjunction of data terms $o[m \twoheadrightarrow \{a_1\}]$ & ... & $o[m \twoheadrightarrow \{a_k\}]$.
- A rule containing a conjunction in the head is equivalent to a set of rules with identical bodies with each head being one of the conjuncts. For example,
  $o1[m1 \rightarrow v1; m2 \rightarrow v2] \leftarrow o2[m3 \rightarrow v3]$.
  becomes
  $o1[m1 \rightarrow v1]$ & $o[m2 \rightarrow v2] \leftarrow o2[m3 \rightarrow v3]$.
  which becomes the two rules
  $o1[m1 \rightarrow v1] \leftarrow o2[m3 \rightarrow v3]$.
  $o1[m2 \rightarrow v2] \leftarrow o2[m3 \rightarrow v3]$.

We can now use these equivalences to rewrite any F-logic program into a normal form where each rule contains a single term in the head and a conjunction of literals in the body.

This provides the advantage that no special unification algorithm need be implemented to deal with sets or complex terms. Henceforth, we consider only *normal* F-logic programs, those resulting from the above translations.

3 Overview of the interpreter

Our primary aim was to build a model interpreter which would faithfully and simply capture the semantics of F-logic, allowing us to write and run small F-logic programs. A secondary aim was to gain insight into which features of declarative object-oriented languages were and were not useful, and the difficulty or cost of implementing them. An efficient implementation was deemed to be less important than a simple and clear one.

Our interpreter deals with only a subset of F-logic. These are programs where the head and body of a rule is a conjunction of, possibly negated, terms. In this sense we deal with F-logic as a programming language, rather than as a logic.

We have implemented the interpreter in NU-Prolog [6] since it provides safe implementations of negation and universal quantification. Porting to Quintus or other implementations of Prologs should be straightforward with help from Quintus' `library(foreach)` and `library(not)`.

The interpreter can be easily split into several parts, each part implementing a different feature of F-logic. The minimal interpreter implements only inheritance for single-valued methods. However, it does not enforce that methods are in fact single-valued. The extensions, in turn, add overriding and enforce single-valuedness; set-valued methods with overriding; signature terms and typing; and p-terms and negation.

We chose the following Prolog representation for F-logic atoms because any part of an F-logic atom may be non-ground. In keeping with the semantics of F-logic we separated the representation of each kind of term. The choice of f_I, etc. is

historical and stems, in part, from the semantic structure of an F-logic program (see [3, page 13]).

F-logic term	Prolog representation
isa-term $X : Y$	$isa(X, Y)$
functional data-term $O[M@A_1, \ldots, A_k \rightarrow V]$	$f\_I(O, M, k, [A_1, \ldots, A_k], V)$
set-valued data-term $O[M@A_1, \ldots, A_k \twoheadrightarrow \{V\}]$	$f\_Iset(O, M, k, [A_1, \ldots, A_k], V)$
functional signature-term $O[M@A_1, \ldots, A_k \Rightarrow \{V\}]$	$f\_II(O, M, k, [A_1, \ldots, A_k], V)$
set-valued signature-term $O[M@A_1, \ldots, A_k \Rightarrow\!\!\!\Rightarrow \{V\}]$	$f\_IIset(O, M, k, [A_1, \ldots, A_k], V)$
p-term $f(A_1, \ldots, A_k)$	$p\_tuple(f(A_1, \ldots, A_k))$
rules $H.$ $H \leftarrow B_1$ & ... & $B_k.$	$rule(H, [])$ $rule(H, [B_1, \ldots, B_k])$

For convenience of manipulation in the interpreter, the bodies of clauses are represented as a list of terms. The empty body is represented by the empty list.

We now have a representation of F-logic programs as a set of rules, each with a single term in the head and a conjunction of literals in the body. A *possible computed answer* for a term T and a program P is an instance $T\theta$ of T such that there exists an instance $T\theta \leftarrow B\theta$ of a clause in P such that the interpreter succeeds when executed with the goal $\leftarrow solve(B\theta)$ and program P. A possible computed answer may or may not belong to the set of *computed answers*, after overriding has been taken into account. Because the rule which produces a possible computed answer may have been inherited, we need to know on which class the rule was defined, to later determine whether the possible computed answer should be overridden or not. We call this class the *source* class.

4 The minimal interpreter

Figure 1 shows the Prolog code for a minimal interpreter implementing inheritance but not sets, typing, overriding, negation or functionality (single-valuedness) of methods.

The interpreter is similar to the standard Prolog meta-interpreter. The predicate `solve_term` handles each of the different kinds of term. In the minimal interpreter we deal only with isa-terms and data-terms.

The predicates `direct_subclass` and `subclass` together compute the inheritance hierarchy where the first argument is the object or class, and the second is the superclass.

The predicate `direct_pca` takes a goal, finds a matching clause and evaluates the

```
solve([]).
solve([H |T]) :- solve_term(H), solve(T).

solve_term(isa(O,C)).
solve_term(isa(O,C)) :- subclass(O, C).
solve_term(f_I(O,M,K,Args,V)) :- pca(O, f_I(C,M,K,Args,V), C).

subclass(O, C) :- direct_subclass(O, C).
subclass(O, C) :- direct_subclass(O, C1),
                  subclass(C1, C).

direct_subclass(O, C) :- rule(isa(O,C), Body), solve(Body).

pca(Obj, Goal, Obj)   :- direct_pca(Goal).
pca(Obj, Goal, Class) :- subclass(Obj, Class),
                         direct_pca(Goal).

direct_pca(Goal) :- rule(Goal, Body), solve(Body).
```

Figure 1: Minimal interpreter – no sets, typing, overriding, or negation.

corresponding body. The predicate `pca` computes the possible computed answers for us. It takes as inputs the class to begin searching for an answer, the goal to solve and returns the source class where the possible computed answer was found.

The recursive calls to `solve` in `direct_subclass` indicates that the inheritance hierarchy can be conditionally defined. This is one reason why it is difficult to compile out some of the inheritance of methods as some languages do.

For example, consider the following program.

$a : b.$
$a : c \leftarrow d[m \rightarrow v1].$
$d : e.$
$b[m1 \rightarrow v2].$
$e[m \rightarrow v3]$

To evaluate the query $a[m1 \rightarrow V]$ we need to know whether a inherits from c but this cannot be determined without evaluating $d[m \rightarrow v1]$.

There is already great potential for our interpreter to loop infinitely due to the dynamic nature of the inheritance hierarchy. Executing the interpreter under a tabling Prolog interpreter [5, 7] would alleviate these problems, but would also introduce new ones as we will see later.

5 Extending the interpreter

As noted above, the interpreter presented so far lacks most of the features one associates with an object-oriented language, other than inheritance of methods. We now present a series of extensions to the simple interpreter, adding overriding and functional methods, sets, typing, p-terms and negation.

5.1 Overriding and functional methods

We require two properties of functional methods which our interpreter does not yet enforce. The first is that they be single-valued. The second is that a method definition lower in the inheritance hierarchy overrides any definitions for the same method higher in the inheritance hierarchy. We can state these properties more precisely as follows.

> *A possible computed answer is a computed answer if and only if all other (different) possible computed answers come from higher in the hierarchy.*

In our interpreter `pca` returns each possible computed answer along with the class from which the answer was derived. Hence, we can code the above principle by replacing the last clause of `solve_term` with the following.

```
solve_term(f_I(O,M,K,Args,V)) :-
        pca(O, f_I(C,M,K,Args,V), C),
        all( [C1, V1],
             ( pca(O, f_I(C1,M,K,Args,V1), C1) =>
                 ( subclass(C, C1) ; V = V1 )
             )).
```

Here, we still call `pca` to find a possible computed answer, but then we check that, for all other possible computed answers they either come from further up in the hierarchy (and are thus overridden) or they are the same as this possible computed answer (which enforces the method to be single-valued). If there is more than one possible computed answer at the lowest level then no answer is returned.

The meta-predicate `all/2` is a NU-Prolog builtin, where the variables in the first argument are universally quantified in the second argument. The Quintus Prolog libraries provide alternatives which can be used to implement `all/2`.

This is perhaps the most important extension to our interpreter as it captures one of the essential aspects of an object-oriented language.

Of course this is not a very efficient implementation since once we have an answer (for a given object, method, and arguments), there can be no others. This code will backtrack unnecessarily through all the possible computed answers. However, it has the advantage of being a simple statement of the actual semantics of functional methods and overriding in F-logic and can be used as a reference point when attempting a more efficient implementation.

Unfortunately the semantics of overriding given here does not match those of F-logic under certain circumstances. These cases occur when overriding and inheritance

depend on each other. For example,

$a : p.$
$a : t \leftarrow a[m \rightarrow 3].$
$p[m \rightarrow 3].$
$t[m \rightarrow 5].$

In this case, there are two models in F-logic, one containing $a[m \rightarrow 3]$ and the other containing $a[m \rightarrow 5]$. This is not very satisfactory. In our interpreter running under XOLDTNF, which gives a three-valued well-founded semantics, $a[m \rightarrow X]$ would be undefined, which fits better with our intuition.

However, in the majority of cases overriding and inheritance do not interfere with each other and our implementation gives the correct semantics. It is desirable to be able to characterise these cases, and this is the subject of ongoing research. We say more about this in section 6.

5.2 Sets

Overriding of set-valued methods in F-logic is one of the more confusing aspects of its semantics. This is because a set-valued method for a class only overrides the method defined for its superclass(es) if it is not a subset of that method.

For example, given the statements $c1[m \twoheadrightarrow \{a, b\}]$, $c2[m \twoheadrightarrow \{a\}]$, and $c2 : c1$, then $c2[m \twoheadrightarrow \{b\}]$ also holds. This is because a statement such as $c2[m \twoheadrightarrow \{a\}]$ only states that a is among the set of values of method m applied to class $c2$. On the other hand, given the statements $c1[m \twoheadrightarrow \{a, b\}]$, $c2[m \twoheadrightarrow \{a, c\}]$, and $c2 : c1$, then $c2[m \twoheadrightarrow \{b\}]$ does not hold because $\{a, c\}$ is not a subset of $\{a, b\}$.

The consequence of this is that, for set-valued methods, we need to evaluate the method at every level of the inheritance hierarchy to check whether any overriding should occur. We can state this more precisely as follows.

> *Given an object o, a possible computed answer with source class c is a computed answer if and only if, for every class c' such that $o \leq c' \leq c$, the possible computed answers with source class c are a superset of the possible computed answers with source class c'.*

We encode these semantics in the following piece of code.

```
solve_term(f_Iset(O,M,K,Args,V)) :-
        pca(O, f_Iset(C,M,K,Args,V), C),
        all( [V1, C1],
             ( (subclass(C1, C),
                pca(O, f_Iset(C1,M,K,Args,V1), C1)) =>
                    pca(O, f_Iset(C,M,K,Args,V1), C)
             )).
```

This is also not very efficient, as it involves multiple traversals of the hierarchy and computes the same answers many times. An equivalent but more efficient method is to traverse the hierarchy bottom-up, stopping as soon as we discover a possible

computed answer which is not a computed answer for an immediate superclass. If the possible computed answers at this level do form a subset of the computed answers of the next level up then we return all the computed answers from the next level up which are not answers at this level, thus removing duplicates. This more procedural implementation of the semantics is illustrated in the following code.

```
solve_term(f_Iset(O,M,K,Args,V)) :-
        direct_pca(f_Iset(O,M,K,Args,V)).
solve_term(f_Iset(O,M,K,Args,V)) :-
        direct_subclass(O, C),
        all( [V1],
             ( direct_pca(f_Iset(O,M,K,Args,V1)) =>
                 solve([f_Iset(C,M,K,Args,V1)])
             )),
        solve([f_Iset(C,M,K,Args,V)]),
        not direct_pca(f_Iset(O,M,K,Args,V)).
```

The first clause says that every possible computed answer at the bottom level is a computed answer. The second clause says that, if the possible computed answers at the bottom level are a subset of the computed answers of an immediate superclass then we inherit all the (different) computed answers from that immediate superclass.

Note that, to get all solutions to a query like $cs\_dept[employees \twoheadrightarrow X]$, we must backtrack through them as X will only be bound to one answer at a time.

5.3 Typing

In F-logic the type of a method is set-valued and always accumulates. That is, types can never be overridden so the value(s) of a method must belong to all the method's types. To find the types of a method we add the following clauses.

```
solve_term(f_II(O,M,K,Args,V)) :-
        pca(O, f_II(C,M,K,Args,V), C).
solve_term(f_IIset(O,M,K,Args,V)) :-
        pca(O, f_IIset(C,M,K,Args,V), C).
```

Once we know the type(s) of a method, we can filter out any possible computed answers which violate the typing constraint. In F-logic a possible computed answer for a method must belong to every type of that method. To ensure this we replace the definition of `direct_pca` with the following clause.

```
direct_pca(Goal) :-
        rule(Goal, Body),
        solve(Body),
        well_typed(Goal).
```

The definition of `well_typed` is as follows. A consequence of this definition is that, if no type information exists for a method, then `well_typed` will always succeed for that method since the left-hand side of the implication will be false.

```
well_typed(f_I(O,M,K,Args,Val)) :-
        all( [Type |ArgTypes],
             ( solve([f_II(O,M,K,ArgTypes,Type)]) =>
                 ( has_type(Val, Type),
                   has_types(Args, ArgTypes)
                 )
             )).
well_typed(f_Iset(O,M,K,Args,Val)) :-
        all( [Type |ArgTypes],
             ( solve([f_IIset(O,M,K,ArgTypes,Type)]) =>
                 ( has_type(Val, Type),
                   has_types(Args, ArgTypes)
                 )
             )).
well_typed(f_II(_,_,_,_,_)).
well_typed(f_IIset(_,_,_,_,_)).

has_types([], []).
has_types([Arg |Args], [Type |Types]) :-
        has_type(Arg, Type),
        has_types(Args, Types).

has_type(Val, Type) :- solve([isa(Val,Type)]).
```

Note that we also check that the arguments to methods are well-typed, and that signature-terms do not require type checking. In an actual implementation, we would expect an error to be reported if `has_type` fails. The appendix contains an appropriate definition of `has_type` for this.

5.4 P-Terms and negation

As described in the F-logic paper [3], p-terms can be treated as syntactic sugar. However, we choose to implement them explicitly to improve efficiency and to facilitate access to certain builtin predicates.

The following clause suffices.

```
solve_term(p_tuple(Term)) :- direct_pca(p_tuple(Term)).
```

To implement negation we rely on the standard negation as failure rule of the underlying Prolog system. Since our system is built on top of NU-Prolog, we can safely add just the following clause, where `Goals` is a list.

```
solve_term(not(Goals)) :- not solve(Goals).
```

6 Lessons learned

We have presented a short, clear interpreter for a subset of F-logic. The simplicity of the interpreter clarifies the logic of overriding as defined in the F-logic paper. It also

clarifies issues in defining the semantics of declarative object-oriented languages and has guided us in the development of our colleagues' language Gulog [2, 1].

Since F-logic clause heads can be isa-terms or signature-terms and their bodies can contain a mixture of isa-terms, data-terms, and signature-terms, we cannot separate the inheritance hierarchy or the typing of methods from the normal evaluation process. This means that, for an F-logic program to be intelligible, some kind of stratification condition on programs is required; otherwise, the tangled interaction of inheritance, type checking, and overriding are effectively impossible to understand. This need is also suggested by the use of negation (universal quantification) in the interpreter.

Our interpreter departs from the F-logic semantics in the difficult cases where inheritance, deduction, and overriding all interact. In these cases (see [3, Appendix B]), the choice of preferred model is somewhat arbitrary anyway.

A second problem is that possible computed answers are unnecessarily computed. For example, for functional methods, there is no point in computing further possible computed answers once a computed answer has been found, since they will be overridden. This is a problem for two reasons. First, in the presence of function symbols, there may be an infinite number of possible computed answers, which are all overridden. In this implementation all possible computed answers will be computed as part of the overriding check, so the interpreter will loop infinitely. A similar problem occurs when function symbols are used to generate an infinite number of types. Second, to extend the interpreter to handle updates or methods with side-effects, we need to ensure that overridden methods are not evaluated and that other methods are evaluated at most once (per invocation).

Function symbols also present a problem when running the interpreter with a tabling mechanism such as XOLDTNF [7] since, in all but the simplest cases, infinite domains are generated. For example, consider the following two clauses defining a generic list type.

$$
\begin{aligned}
&nil : list(X).\\
&cons(H,T) : list(X) \leftarrow\\
&\qquad H : X \ \&\ T : list(X).
\end{aligned}
$$

The simple query $nil : list(nil)$ can cause XOLDTNF to try to enumerate the entire domain (since $nil : list(list(nil))$, $nil : list(list(list(nil)))$, etc.).

Finally, it would be easier to optimise the evaluation of a query to avoid unnecessary computation if all the variables were typed. For example, the rule

$$X[name \rightarrow bar] \leftarrow X : foo \ \&\ rest.$$

only ever applies to objects of class foo, yet it will unify with every method call to $name$ (with zero arguments).

This leads us to the following conclusions regarding F-logic and its suitability as a basis for an object-oriented logic programming language. The set-valued method overriding semantics are too complex and necessarily inefficient to implement. A better semantics would be something closer to those for single-valued methods. This would also be more intuitive for the user.

Typing should probably be made more explicit by requiring all variables to be typed. Prohibiting isa-terms and data-terms from clauses defining method signatures, and data and signature terms from clauses defining the inheritance hierarchy is

probably too restrictive, but imposing a stratification condition with respect to these clauses would probably help. The j-stratification condition of Gulog [2] seems like a reasonable restriction to impose without sacrificing the expressiveness of the language too much. It closely corresponds to a modular stratification condition on the Prolog representation of F-logic programs with respect to our interpreter.

Future work will involve designing a new language which extends a subset of F-logic to support updates and, in general, methods with side-effects. This is not a trivial task since we need to avoid executing methods with side-effects more than once and we don't want to execute them at all if they are overridden.

We plan to implement this language on top of a deductive database, and to include mechanisms for specifying and checking integrity constraints.

References

1. G. Dobbie and R. W. Topor. A model for inheritance and overriding in deductive object-oriented systems. In G. Gupta, G. Mohay, and R. W. Topor, editors, *Proc. of the 16th Australian Computer Science Conference*, volume 15, pages 625–634, Brisbane, Queensland, Feb. 1993.

2. G. Dobbie and R. W. Topor. Representing inheritance and overriding in Datalog. Unpublished report, 1993.

3. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical report 90/14 (revised), Department of Computer Science, State University of New York at Stony Brook, Aug. 1990.

4. A. Lefebvre, P. Bernus, and R. W. Topor. Querying heterogeneous databases: A case study. In M. Orlowska and M. Papazoglou, editors, *Proc. of the 4th Australian Database Conference*, pages 186–197, Brisbane, Australia, Feb. 1993. World Scientific.

5. K. F. Sagonas, T. Swift, and D. S. Warren. *The XSB Programmer's Manual Version 1.0 (β).* Department of Computer Science, State University of New York at Stony Brook, Mar. 1993.

6. J. Thom and J. Zobel. NU-Prolog reference manual, version 1.5.24. Technical Report 86/10, Department of Computer Science, University of Melbourne, 1990.

7. D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, Mar. 1992.

Appendix

```
solve([]).
solve([H |T]) :- solve_term(H), solve(T).

solve_term(not(Goals)) :- not solve(Goals).
solve_term(p_tuple(Term)) :- direct_pca(p_tuple(Term)).
solve_term(isa(O,O)).
solve_term(isa(O,C)) :- subclass(O, C).
solve_term(f_I(O,M,K,Args,V)) :-
        pca(O, f_I(C,M,K,Args,V), C),
        all( [C1, V1],
             ( pca(O, f_I(C1,M,K,Args,V1), C1) =>
                 ( subclass(C, C1) ; V = V1 )
             )).
solve_term(f_Iset(O,M,K,Args,V)) :-
        direct_pca(f_Iset(O,M,K,Args,V)).
solve_term(f_Iset(O,M,K,Args,V)) :-
        direct_subclass(O, C),
        all( [V1],
             ( direct_pca(f_Iset(O,M,K,Args,V1)) =>
                 solve([f_Iset(C,M,K,Args,V1)])
             )),
        solve([f_Iset(C,M,K,Args,V)]),
        not direct_pca(f_Iset(O,M,K,Args,V)).
solve_term(f_II(O,M,K,Args,V)) :-
        pca(O, f_II(C,M,K,Args,V), C).
solve_term(f_IIset(O,M,K,Args,V)) :-
        pca(O, f_IIset(C,M,K,Args,V), C).

subclass(O, C) :- direct_subclass(O, C).
subclass(O, C) :- direct_subclass(O, C1),
                  subclass(C1, C).

direct_subclass(O, C) :- rule(isa(O,C), Body), solve(Body).

pca(Obj, Goal, Obj)   :- direct_pca(Goal).
pca(Obj, Goal, Class) :- subclass(Obj, Class),
                         direct_pca(Goal).

direct_pca(Goal) :-
        rule(Goal, Body), solve(Body), well_typed(Goal).

well_typed(f_I(O,M,K,Args,Val)) :-
        all( [Type |ArgTypes],
             ( solve([f_II(O,M,K,ArgTypes,Type)]) =>
                 ( has_type(Val, Type),
```

```
                  has_types(Args, ArgTypes)
                )
            )).
well_typed(f_Iset(O,M,K,Args,Val)) :-
        all( [Type |ArgTypes],
             ( solve([f_IIset(O,M,K,ArgTypes,Type)]) =>
                 ( has_type(Val, Type),
                   has_types(Args, ArgTypes)
                 )
             )).
well_typed(f_II(_,_,_,_,_)).
well_typed(f_IIset(_,_,_,_,_)).

has_types([], []).
has_types([Arg |Args], [Type |Types]) :-
        has_type(Arg, Type),
        has_types(Args, Types).

has_type(Val, Type) :-
        ( solve([isa(Val,Type)]) ->
            true
        ;
            format("Type error: ~p not of type ~p.\n", [Val, Type]),
            fail
        ).
```