Indexing `dif/2`

ULRICH NEUMERKEL
TU Wien, Vienna, Austria
(*e-mail:* `ulrich@complang.tuwien.ac.at`)

STEFAN KRAL
FH Wiener Neustadt, Austria
(*e-mail:* `stefan.kral@fhwn.ac.at`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Many Prolog programs are unnecessarily impure because of inadequate means to express syntactic inequality. While the frequently provided built-in `dif/2` is able to correctly describe expected answers, its direct use in programs often leads to overly complex and inefficient definitions — mainly due to the lack of adequate indexing mechanisms. We propose to overcome these problems by using a new predicate that subsumes both equality and inequality via reification. Code complexity is reduced with a monotonic, higher-order if-then-else construct based on `call/N`. For comparable correct uses of impure definitions, our approach is as determinate and similarly efficient as its impure counterparts.

KEYWORDS: Prolog, monotonicity, inequality, reification

1 Introduction

Do Prolog programmers really have to choose between logical purity and efficiency? Even for the most elementary notion of syntactic equality this question still remains unanswered. Today, many Prolog programs consist of unnecessarily procedural constructs that have been motivated by efficiency considerations blurring the declarative vision. To improve upon this situation we need pure constructs that are of comparable efficiency as their impure counterparts. We focus our attention on the pure, monotonic subset of modern Prolog processors. The monotonic subset has many desirable properties: it fits seamlessly with constraints, enables declarative debugging and program slicing techniques (Weiser 1982; Neumerkel and Kral 2002), and is directly compatible with alternative search procedures like iterative deepening. Our effort aims into the same direction that Functional Programming took so successfully; away from a command-oriented view to the pure core of the paradigm.

Many recent developments have facilitated pure programming in Prolog. In particular, the widespread adoption of the higher-order built-in predicate `call/N` (O'Keefe 1984) together with its codification (ISO/IEC 13211-1:1995/Cor.2 2012) has paved the way to yet unexplored pure programming techniques. On another track, more and more Prolog processors are rediscovering the virtues of syntactic inequality via `dif/2`.

The very first Prolog, sometimes called Prolog 0 (Colmerauer et al. 1973) already supported `dif/2`. Unfortunately, the popular reimplementation Prolog I (Battani and Meloni 1973) omitted `dif/2` and other coroutining features. This system was the basis for Edinburgh Prolog (Pereira

et al. 1978) which led to ISO-Prolog (ISO/IEC 13211-1 1995). After Prolog I, `dif/2` was reintroduced in Prolog II; independently reinvented in MU-Prolog (Naish 1986) and soon implementation schemes to integrate `dif/2` and coroutining into efficient systems appeared (Carlsson 1987; Neumerkel 1990). The major achievement was that the efficiency of general Prolog programs *not* using `dif/2` remained unaffected within a system supporting `dif/2`. In this manner `dif/2` survived in major high-performance implementations like SICStus. However, it still has not gained general acceptance among programmers. We believe that the main reason for this lack of acceptance is that `dif/2` does not directly deliver the abstraction that is actually needed. Its direct use leads to clumsy and unnecessarily inefficient code. Its combination with established control constructs often leads to unsound results. New, pure constructs are badly needed.

Contents. We first recall the deficiencies of Prolog's if-then-else control constructs. Then the hidden deficiencies of the pure definition of `member/2` are exposed. A refined version is given whose efficiency is subsequently improved with the help of reification and a new, pure and monotonic if-then-else construct. Finally, we show how our approach permits to define more complex cases of reification and compare it to constructive negation.

2 The declarative limits of Prolog's if-then-else

Prolog's if-then-else construct was first implemented in the interpreter of DEC10 Prolog around 1978 (Pereira et al. 1978); its compiler, however, did not support it. Subsequent implementations, starting with C-Prolog and Quintus Prolog, adopted it fully which led to its inclusion into the ISO standard.

For many uses, this construct provides a clean way to express conditional branching. These uses all assume that the condition is effectively non-recursive and *sufficiently instantiated* to permit a simple test. Some built-in predicates ensure their safe usage by issuing instantiation errors in cases that are too general. For example, the built-in predicates for arithmetic evaluation and comparison like `(is)/2` and `(>)/2` issue instantiation errors according to the general scheme for errors (Neumerkel and Triska 2009). But in general, problems arise. For its common use, the construct `( If_0 -> Then_0 ; Else_0 )` contains three regular goals which is equivalent to `( once(If_0) -> Then_0 ; Else_0 )`. The first answer of `If_0` is taken, and all subsequent answers are discarded. The if-then-else has thus similar problems as a commit operator. And even the "soft cut"-versions `if/3` and `(*->)/2` of SICStus and SWI respectively, expose the same problems as Prolog's unsound negation. MU-Prolog (Naish 1986) provided an implementation of if-then-else that delays the goal `If_0` until it is ground. While sound, such an implementation leads to many answers with unnecessarily floundering goals. Consider the goal `[] = [E|Es]` which is not ground and thus leads to floundering. Even for the cases where this construct works as expected, we still suffer from the lack of monotonicity.

3 What's wrong with `member/2`?

Already pure definitions expose problematic behaviors that ultimately lead to impure code. Consider `member/2`:

`member(X, L)` is true if `X` is an element of the list `L`.

The common actual definition is slightly more general than above since `L` is not required to be a list. Certain instances of partial lists like in the goal `member(a, [a|non_list])` succeed as

well. Such generalizations are motivated by efficiency reasons; the cost for visiting the entire list to ensure its well-formedness is often not acceptable. The complete definition of `member/2` can thus be described as:

`member(X, L)` is true iff `X` is an element of a list prefix of `L`.

```
member(X, [X|_Es]).
member(X, [_E|Es]) :-
   member(X, Es).
```

```
?- member(1, [1,2,3,4,5]).
   true
;  false.
```

```
?- member(1, [1,2,1,4,5]).
   true
;  true
;  false.
```

Above, lines starting with `?-` show queries followed by their answers—similar to SWI's top level shell. Alternative answers are separated by `;`. An alternative answer `; false.` indicates that Prolog needed further computation to ensure that no further answer is present. It is thus an indication that Prolog still uses space for this query, even though no further answer exists. This is thus a source of inefficiency we will address in this paper.

For its first answer the goal `member(1, [1,2,3,4,5])` does not visit the entire list. Nevertheless, upon backtracking, the entire list gets visited anyway. Thus, the well-meant generalization does not lead to a more efficient implementation. For many goals with only a single solution, `member/2` leaves a choicepoint open that can only be reclaimed upon failure or with non-declarative means like the cut. So while `member/2` is itself a pure definition, its space consumption forces a programmer to resort to impurity. A common library predicate to this end is `memberchk/2` which does not leave any choicepoint open at the expense of incompleteness. The precise circumstances when this predicate is safe to use are difficult to describe. Many manuals suggest that the goal needs to be *sufficiently instantiated* without giving a precise criterion. To err on the safe side, cautious programmers need to add tests which are themselves prone to programming errors and incur runtime overheads. In the following example an insufficiently instantiated goal leads to an unexpected failure.

```
memberchk(X, Es) :-
   once(member(X, Es)).
```

```
?- X = 2, memberchk(X, [1,2]), X = 2.
   X = 2.    % expected solution

?-        memberchk(X, [1,2]), X = 2.
   false.    % unexpected failure
```

4 A refurbished `member/2`

The definition of `member/2` contains unnecessary redundancy. This becomes apparent when rewriting the two clauses to an explicit disjunction. In the first branch `X = E` holds, but in the second branch this may hold as well. This can be observed with the query `member(1, [1,X])` shown below. The second answer `X = 1` is already subsumed by the first answer `true`. The branches of the disjunction are thus not mutually exclusive. By adding an explicit `dif/2`[1] to the

[1] An ISO conforming definition is given in the appendix for systems without `dif/2`.

second branch this redundancy is eliminated. Note that there are still possibilities for less-than-optimal answers as in the query `memberd(1, [X,1])` where the two answers could be merged into a single answer.

```
member(X, [E|Es]) :-
   (  X = E
   ;  member(X, Es)
   ).

?- member(1, [1,X]).
   true
;  X = 1. % redundant answer

?- member(1, [X,1]).
   X = 1
;  true. % ~ redundant
```

```
memberd(X, [E|Es]) :-
   (  X = E
   ;  dif(X, E),
      memberd(X, Es)
   ).

?- memberd(1, [1,X]).
   true
;  false.

?- memberd(1, [X,1]).
   X = 1
;  dif(X, 1)
;  false.

?- memberd(1, [1,2,3]).
   true
;  false. % leftover choicepoint
```

For sufficiently instantiated cases where `memberchk/2` yields correct results, there are no redundant answers for `memberd/2`. However, it still produces "leftover choicepoints" displayed as `; false`. Space is thus consumed, even after succeeding. This is a frequent problem when using `dif/2` directly: it cannot help to improve indexing since it is implemented as a separate built-in predicate. Indexing techniques have been developed both for the rapid selection of matching clauses and to prevent the creation of superfluous choicepoints. They are even more essential to pure Prolog programs which cannot resort to impure constructs like the cut. With `dif/2` the situation is similar: for the frequent case that `X` and `E` are identical, a choicepoint is created even though we know that the goal `dif(X, E)` will fail upon backtracking. Further, programming with `dif/2` is rather cumbersome since all conditions have to be stated twice: once for the positive case and once for the negative. Therefore, for both execution and programmer efficiency, a new formulation is needed.

5 Reification of equality

The disjunction `X = E ; dif(X, E)` is combined into a new predicate `=(X, E, T)` with an additional argument which is `true` if the terms are equal and `false` otherwise. In this manner the truth value is reified. An implementer is now free to replace the definition of `(=)/3` by a more efficient version. The simple ISO conforming implementation in the appendix is already able to eliminate many unnecessary choicepoints for all cases where the terms are either identical or not unifiable. A more elaborate implementation might avoid to visit the terms several times.

```
=(X, X, true).
=(X, Y, false) :-
   dif(X, Y).
```

```
memberd(X, [E|Es]) :-
   =(X, E, T),
   (  T = true
   ;  T = false,
      memberd(X, Es)
   ).
```

Still, this direct usage of reifying predicates does not address all our concerns. On the one hand there is an auxiliary variable for each reified goal and on the other hand many Prolog implementations cannot perform the above disjunction without a leftover choicepoint. Both issues are addressed using a higher-order predicate.

6 The monotonic `if_/3`

Our new, monotonic if-then-else is of the form `if_(If_1, Then_0, Else_0)`. The condition `If_1` is now no longer a goal but an incomplete goal, which lacks one further argument. That argument is used for the reified Boolean truth value.

```
memberd(X, [E|Es]) :-
   if_( X = E       % (=)/3
      , true
      , memberd(X, Es)
      ).
```

```
?- memberd(1, [1,X]).
   true.   % fully deterministic

?- memberd(1, [1,2,3]).
   true.   % fully deterministic
```

The implementation of `if_/3` given in the appendix already avoids many useless choicepoints. Our choice to use a ternary predicate in place of the nested binary operators was primarily motivated by the semantic difficulties in ISO Prolog's if-then-else construct whose principal functor is `(;)/2` and not `(->)/2`. This means that there are two entirely different control constructs with the very same principal functor: 7.8.6 `(;)/2` – disjunction and 7.8.8 `(;)/2` – if-then-else (ISO/IEC 13211-1 1995). To avoid this very hard-to-resolve ambiguity, we chose `if_/3`.

While our implementation of `if_/3` in the appendix avoids the creation of many useless choicepoints, the overall performance relies heavily on the meta-call `call/N`. For this reason, we provide an expanding version that removes many meta-calls in `if_/3`. This goal expansion is provided in `library(reif)`. Table 1 compares two uses of `memberchk/2` with their pure counterparts. First, the letter 'z' is searched in a list with all letters from a to z followed by a space. The second test searches in a list of pairs of the form 'Key-Value' the 10th element via a key (`lassoc`). Note that the impure and pure version are slightly different. The impure version may skip invalid elements that are not pairs ; the pure version will fail in such a case. The built-in `memberchk/2` and a version based on `once(member(X,Xs))` are compared with pure and complete versions, either using `dif/2` directly ; or using `if_/3` directly ; as well as the expansion generated by `library(reif)`. The measurements were performed with 10^6 repetitions of the goals on an Intel Core i7-4700MQ 2.4 GHz using SICStus 4.3.2 and SWI 7.3.20. Using `-O` for SWI did not produce any improvements. We believe that the current overheads of a factor of 2 to 3 could be further reduced if specialized built-ins for conditional testing and a better register allocation scheme in SICStus were available.

Based on `if_/3`, many idiomatic higher-order constructs can be defined, now with substantially more general uses. The commonly used predicate for filtering elements of a list, often called

Table 1. *Runtimes of impure vs. pure definitions*

	`memberchk/2`		`memberd/2`		
system	built-in	once	dif	if_/3	expanded
SICStus	0.690s	0.690s	1.620s	7.050s	1.380s
SWI	0.761s	1.890s	8.493s	10.053s	2.463s

	`lassoc/3`		`memberk/3`		
system	built-in	once	—	if_/3	expanded
SICStus	0.470s	0.470s		3.060s	0.640s
SWI	0.591s	1.140s		4.641s	1.350s

`include/3` or `filter/3`, is now replaced by a definition `tfilter/3` using a reified condition. The first argument of `tfilter/3` is an incomplete goal which lacks two further arguments. One for the element to be considered and one for the truth value. The following queries illustrate general uses of this predicate that cannot be obtained with the traditional definitions.

```
tfilter(_CT_2,    [], []).
tfilter(CT_2, [E|Es], Fs0) :-
   if_(call(CT_2,E), Fs0 = [E|Fs], Fs0 = Fs ),
   tfilter(CT_2, Es, Fs).

?- tfilter(=(X), [1,2,3,2,3,3], Fs).
   X = 1, Fs = [1]
;  X = 2, Fs = [2,2]
;  X = 3, Fs = [3,3,3]
;  Fs = [], dif(X, 1), dif(X, 2), dif(X, 3).

duplicate(X, Xs) :-
   tfilter(=(X), Xs, [_,_|_]).
```

```
?- duplicate(X, [1,2,3,2,3,3]).
   X = 2
;  X = 3
;  false.
```

7 General reification

So far, we have only considered the reified term equality predicate `(=)/3`. Each new condition of `if_/3` requires a new reified definition. In contrast to constructive negation (Chan 1989; Drabent 1995), these definitions are not constructed automatically. To test for membership the following reified definition might be used.

```
memberd_t(X, Es, true) :-
   memberd(X, Es).
memberd_t(X, Es, false) :-
   maplist(dif(X), Es).
```

This definition insists on a well-formed list in the negative case. Contrast this to constructive negation which considers any failing goal `memberd(X, Es)` as a valid negative case. Our definition thus fails for `memberd_t(X, non_list, T)`, that is, this case is neither true nor false. Constructive negation would consider this a case for `T = false`. On the other extreme are approaches that guarantee that type restrictions are maintained. However, our definition happens to be true for `memberd_t(1, [1|non_list], true)` — for efficiency reasons. A system maintaining the list-type would fail in this case whereas we visit the list only for the very necessary parts. It is this freedom between the two extremes of constructive negation and well-typed restrictions that permits an efficient implementation of above definition. In fact, the following improved definition visits lists in the very same manner as the impure `memberchk/2` — for comparable cases. And for more general cases it maintains correct answers.

```
memberd_t(X, Es, T) :-
   l_memberd_t(Es, X, T).

l_memberd_t([], _, false).
l_memberd_t([E|Es], X, T) :-
   if_( X = E
      , T = true
      , l_memberd_t(Es, X, T) ).
```

```
firstduplicate(X, [E|Es]) :-
   if_( memberd_t(E, Es)
      , X = E
      , firstduplicate(X, Es)
      ).
```

```
?- firstduplicate(1, [1,2,3,1]).
   true.

?- firstduplicate(X, [1,2,2,1]).
   X = 1.
```

```
?- firstduplicate(X, [A,B,C]).
   X = A, A = B
;  X = A, A = C, dif(C, B)
;  X = B, B = C, dif(A, C), dif(A, C)
;  false.
```

The following example shows how we deal with reification in the general case. Again, it guarantees a well-typed tree only for the negative case. In the positive case, the tree is visited only partially. Note that the reified version on the right uses a reified version of disjunction. Instead of `(;)/2` the reified `(;)/3` defined in the appendix is used. This reified version is now significantly more compact than defining the positive and negative cases explicitly. In fact, it is close in size to the positive case (`treemember/2`) alone.

```
treemember_t(E, Tr, true) :-
   treemember(E, Tr).
treemember_t(E, Tr, false) :-
   tree_non_member(E, Tr).

treemember(E, t(F,L,R)) :-
   (  E = F
   ;  treemember(E, L)
   ;  treemember(E, R)
   ).

tree_non_member(_, nil).
tree_non_member(E, t(F,L,R)) :-
   dif(E, F),
   tree_non_member(E, L),
   tree_non_member(E, R).
```

```
treememberd_t(_, nil, false).
treememberd_t(E, t(F,L,R), T) :-
   call(
     (  E = F
     ;  treememberd_t(E, L)
     ;  treememberd_t(E, R)
     ),
   T).
```

8 Conclusion

We have presented a new approach to improving the efficiency of pure programs using syntactic inequality. Our solution to indexing `dif/2` was to provide a generalized reifying definition together with a monotonic if-then-else construct. In this manner the calls to the actual built-in `dif/2` are reduced to those cases that actually need its general functionality. In all other situations, the programs run efficiently without calling `dif/2` and without creating many unnecessary choicepoints.

Acknowledgements. The presented programs were publicly developed on comp.lang.prolog and as answers to questions on stackoverflow.com.

2009-10-15 ISO-`dif/2` comp.lang.prolog
2012-12-01 Reification of term equality stackoverflow.com/q/13664870
2014-02-23 `memberd/2` stackoverflow.com/a/21971885
2014-02-23 `tfilter/3` stackoverflow.com/a/22053194
2014-12-09 `if_/3` stackoverflow.com/a/27358600

Further examples: `stackoverflow.com/search?q=[prolog]+if_`

Appendix A Appendix

The full `library(reif)` and the benchmarks (`memberbench`) are available at:
`http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/sicstus`

```
dif(X, Y) :-
   X \== Y,
   ( X \= Y -> true ; throw(error(instantiation_error,_)) ).
```

```
% :- meta_predicate(if_(1, 0, 0)).
if_(If_1, Then_0, Else_0) :-
   call(If_1, T),
   (  T == true -> call(Then_0)
   ;  T == false -> call(Else_0)
   ;  nonvar(T) -> throw(error(type_error(boolean,T),_))
   ;  /* var(T) */ throw(error(instantiation_error,_))
   ).

=(X, Y, T) :-
   (  X == Y -> T = true
   ;  X \= Y -> T = false
   ;  T = true, X = Y
   ;  T = false,
      dif(X, Y)                              % ISO extension
      % throw(error(instantiation_error,_)) % ISO strict
   ).

','(A_1, B_1, T) :-
   if_(A_1, call(B_1, T), T = false).

;(A_1, B_1, T) :-
   if_(A_1, T = true, call(B_1, T)).
```

References

G. Battani, H. Meloni. Interpréteur du langage de programmation Prolog, Rapport de D.E.A. d'Informatique Appliquée. Groupe d'Intelligence Artificielle, U.E.R. de Luminy. Université d'Aix-Marseille. 1973.

M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. ICLP 1987.

D. Chan. An Extension of Constructive Negation and its Application in Coroutining. NACLP 1989.

A. Colmerauer, H. Kanoui, Ph. Roussel, R. Pasero. Un système de communication homme-machine en Français, Rapport de recherche, CRI 72-18. U.E.R de Luminy. Université d'Aix-Marseille. 1972-1973.

W. Drabent. What is failure? An approach to constructive negation. Acta Informatica 32(1):27-59, 1995.

ISO/IEC 13211-1:1995 Programming languages - Prolog - Part 1: General core.

ISO/IEC 13211-1:1995/Cor 2:2012. Second Technical Corrigendum for Programming languages - Prolog - Part 1: General core.

R. O'Keefe. Draft Proposed Standard for Prolog Evaluable Predicates. 1984.
Copy: http://www.complang.tuwien.ac.at/ulrich/iso-prolog/okeefe.txt

L. Naish. Negation and Control in Prolog. LNCS 238. 1986.

U. Neumerkel. Extensible Unification by Metastructures. META'90. 1990.

U. Neumerkel, St. Kral. Declarative program development in Prolog with GUPU. 12th Workshop on Logic Programming Environments (WLPE), Copenhagen 2002.

U. Neumerkel, M. Triska. An error class for unexpected instantiations. ISO/IEC JTC1 SC22 WG17 N213 2009 and N226 2010. http://www.complang.tuwien.ac.at/ulrich/iso-prolog/error_k

L. M. Pereira, F. C. N. Pereira, D. H. D. Warren. User's Guide to DECsystem-10 Prolog. 1978.

M. Weiser. Programmers Use Slices When Debugging. CACM 25(7): 446-452, 1982.