Published in A. Bossi, editor, *Logic-Based Program Synthesis and Transformation*,

Annotations for Prolog – A Concept and Runtime Handling

M. Kulaš

FernUniversität Hagen, FB Informatik, D-58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract. A concept of annotations for rendering procedural aspects of Prolog is introduced, built around well-known procedural concepts of Standard Prolog. Annotations describe properties of predicates. Such properties can be pre or post conditions, which must hold true when a predicate is called or exited, respectively. We introduce two more kinds of annotations, fail and redo annotations, hence incorporating a whole *model* of Prolog execution into our language. This enables natural rendering of many procedural properties of Prolog which cannot be expressed with only pre/post conditions, like non-failure. Every annotation can be narrowed down to a subset of calls, via *calling premiss*. With the novel idea of *context* we supersede program-point assertions. The annotations are defined simply as Prolog goals, making them fully *parametric* and therefore comfortable for debugging. All examples presented are actual runs of our system Nope.

1 Introduction

This work was motivated by the need for more accurate and more verifiable documentation for Prolog programs. Even in pure logic programs, there is no easy equation "specification = implementation", as pointed out in [11]. All the more so in Prolog, due to the non-logical predicates like cut, var, assert. Unambiguous, terse and still readable comments are hard to write. Perhaps even more difficult is the question, how trustworthy the comment is, i. e. how closely does it match its target? There has been much important research in rendering and verifying aspects of Prolog execution, starting with the introduction of mode declarations. Sec. 7 presents a short survey of related research, biased towards declarative efforts. We indicate along the way what we feel to be missing in the existing approaches. In Sec. 3 a remedy is proposed in the form of call annotations. Sec. 4 is the central part of the paper: it generalizes the concept of call annotations and provides a transformational semantics. Sec. 5 outlines some enhancements of the annotation language. Sec. 6 allows a close look into the merits of the approach, by means of several illustrations. Sec. 8 points out some possible continuations of this work.

2 Modes

While in principle every argument of a predicate may be arbitrarily instantiated, i. e. it can be anything from a variable over terms with variables down to a ground term, in practice there are always restrictions.

To set up the terminology, let us name the degree of instantiation of an invocation of a predicate a *calling pattern*, or call substitution. Similarly, on exit we obtain a corresponding *exit pattern*, or answer substitution. We shall regard a calling pattern and a corresponding exit pattern of a predicate as a *mode* of the predicate, like in the approach of [18]. We say that T is *instantiated* if nonvar(T) holds, and *uninstantiated* if var(T) holds. We say that a term T gets *more instantiated* (gets *changed*) if some variable in T gets instantiated.

The classic mode declarations, introduced by [20] and first provided in DEC-10 Prolog to underpin compiled code optimization, specify the usage of each argument of a predicate by choosing one of the three mode symbols: mode '+' specifies that the corresponding argument in any call of the predicate will always be instantiated (input argument), mode '-' specifies that the argument will always be uninstantiated (output argument), and mode '?' specifies that there is no restriction on the form of the argument.

3 From modes to call annotations

The classic mode language is very simple and useful, but often not expressive enough, the var/nonvar discrimination being too coarse grained for many practical purposes. For example, sometimes we need to specify that an argument has to be ground at the time of call, as in the case of Prolog negation. Namely, if a non-ground negation is called, Prolog might compute an unsound answer ('safe' versus 'floundering' negation).

What would help, in our opinion, is the possibility of expressing the intended calling patterns like this

$$\text{call } \backslash\!+ \text{ X} \Rightarrow \text{ground(X)}$$

to be read as "negated goals shall be ground at the time of call". Let us name such a declaration a *call annotation*. We propose the following syntactic variants.

$$\begin{aligned} \text{call Goal} &\Rightarrow \text{MustHold} \\ \text{call Goal with Template} &\Rightarrow \text{MustHold} \\ \text{call Goal \{ with Template \} within Context} &\Rightarrow \text{MustHold} \end{aligned}$$

We've just seen an example for the first variant, namely the safe negation annotation. There will be diverse examples in this paper, each tested with Nope. Nope is an (almost standard) Prolog module that recognizes our annotations and checks them at run-time.

The second variant is an enhancement of the goal specification in Prolog, namely a template allows us to specify the calling pattern more precisely than with head unifications alone. In fact, arbitrarily precise, since a template can be any Prolog goal. The default template is true.

The third variant introduces the idea of *context*, a way of specifying sets of program points. The contexts are partially ordered (*context subsumption*). The default context is _, which subsumes any other context, so an annotation without a context part applies in any context.

Definition 1 (call annotation). *Let* P *be a predicate of arity* n*. A call annotation for* P/n *is a Prolog goal*

$$\textit{call Premiss} \Rightarrow \textit{Constraint}$$

where Premiss ::= Goal { with Template } { within Context }*. Here* Goal *is a Prolog goal for* P/n*, i. e. a Prolog term of the form* $P(S_1, .., S_n)$*.* Context *is a context (see Sec. 3.3).* Template, Constraint *are arbitrary Prolog goals.*

Technical note: Declarations or goals We show our annotations mostly in the traditional, declaration form :- Annotation. to make them stick out, but please note that they are simply *goals* and can be called like any other Prolog goals, amounting to *parametric* annotations. (In case you wonder which predicate is here being called, it's the arrow.)

Technical note: What may be annotated The annotation language is implemented by means of goal replacement, see page 8. Therefore, in principle there is nothing to prevent us from annotating *arbitrary* goals and not just *atomary* ones (as opposed to composite predications like negation, conjunction, disjunction, or if-then-else). For example, it is just a question of slowing down the parser a notch to allow arbitrary negated goals, making it possible to express safe negation in its most general form (as above). The default parser of our implementation will recognize annotated negation, but conjunction, disjunction and if-then-else will be transparent, i. e. any annotations for these built-in 'predicates' of Standard Prolog will be ignored, as opposed to all other built-ins (and of course all user-defined predicates).

Technical note: Broken annotations in Nope This note shall explain the illustrations. If an annotation does not hold (with respect to the given program and a goal), our runtime checker Nope will issue a warning. The default warning in Nope shows the violated annotation, with its arrow broken. For the safe negation example it would look like

/* NOPE call \+member(A,[1,2]) $\not\Rightarrow$ ground(member(A,[1,2])) */

This can be overridden via an explicit use of the else/2 utility (defined on page 9):

call \+ X $\Rightarrow$ ground(X) else write(floundering)

customizing the warning into /* NOPE floundering */. Also, the warning handler of Nope has different severity levels. For example, the utility bark/0 appearing in Sec. 6.3 tunes the warning handler to interrupt Prolog for each warning.

3.1 Soft success

Owing to the defaults for Template and Context, we may assume every call annotation in the form call Goal with Template within Context $\Rightarrow$ Constraint. There is also a normal form, employed in the implementation of Nope (see page 8).

Definition 2 (normal form). *We say that* call Goal with Template within Context ⇒ Constraint, *where* Goal ≡ $P(S_1, .., S_n)$, *has the normal form* call GoalSkeleton with FullTemplate within Context ⇒ Constraint, *where* GoalSkeleton *is the full generalization of* Goal: GoalSkeleton ≡ $P(X_1, .., X_n)$ *with* $X_1, .., X_n$ *distinct new variables, and all the specializations are made explicit via* FullTemplate ≡ ($X_1 = S_1, .., X_n = S_n$, Template).

An observant reader might object that unifying a goal Q with Goal and computing Template is not the same as unifying Q with GoalSkeleton and computing FullTemplate, see var(3) versus var(X), X=3. But the annotations shall not be applied via unification, since we consider matching (subsumption) to be the more natural choice for our purpose. Recall the traditional concept of Pattern *subsuming* Object if Object is an instance of Pattern, i. e. if Object can be unified with Pattern without further instantiating any variables in Object. We generalize this so as to accommodate not only equality but arbitrary goals in Template: everything goes but there is an invariant, the user's goal, which may not be changed.

Definition 3 (succeeds soft). *Let* Goal *and* Invariant *be arbitrary Prolog goals. We say that* Goal *succeeds soft on* Invariant *if* Goal *succeeds without changing* Invariant.

More formally: If Goal succeeds with an answer substitution σ, then σ may only rename Invariant, written as σ(Invariant) ≅ Invariant.

Soft succeeding goals are not only of the kind \+ \+ G. Nope provides a predicate call_soft(G, J) that succeeds only softly on J for any G and any J. The advantage over \+ \+ G is that soft success allows for *local variables*, because the variables of G not occurring in J may get instantiated.

In Def. 8 we define applicability of annotations via soft success. Owing to this we may have arbitrary premisses, like p(X,Y) with X=Y.

3.2 Simple context

It is useful to be able to access the immediate ancestor (parent) of a goal Q, i. e. the goal which called Q. To this effect, we introduce contexts.

Definition 4 (parent of a goal). *The head of a clause is the parent of each body goal. If a goal is issued at the top-level of a Prolog interpreter, we will say that its parent is the goal* false.

A *context* specifies the parent of the annotated goal. In the simplest case, it can be any atomary Prolog goal. So the meaning of an annotation for Q within Context shall be to check on Q occurring anywhere in the clause bodies of Context. Alternatively, one can view contexts as specifying *sets of program points*, namely those points where the annotated goal is allowed to appear (in this first simple case: all points in the bodies of Context).

Furthermore, sometimes it is useful to exclude contexts. For example, if we only want to check the non-recursive calls of a predicate P, we need to express "check P within any context but P". For such purposes we allow *negative contexts*, represented as dash-prefixed atomary Prolog goals. The meaning of -Context is the complement of the set of program points represented by Context, i. e. this can be any point in the program (plus top-level, 'defining' false/0) outside of the definition of Context.

In the following we shall further refine the idea of a context. But first we have some questions for our contexts so far. One key question about contexts is their inheritance, i. e. when is an annotation for Q within X pertinent to Q within Y? Also, when is an annotation bound to a context CA pertinent to a goal within a context CQ? To settle these questions, we define a partial ordering for contexts.

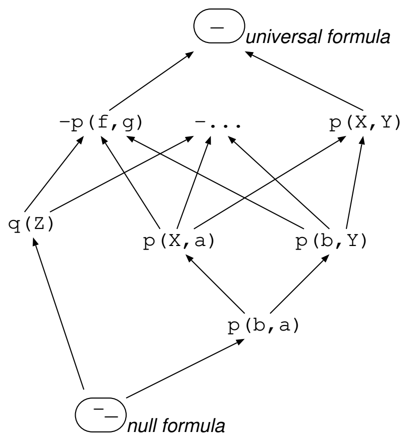

Fig. 1. Poset of simple contexts

We start from the natural lattice of first-order atomary formulas (augmented by a 'universal' and a 'null' formulas) modulo renaming, as introduced by [17]. The partial ordering $\preceq$ being "is instance of", the greatest lower bound of two atoms being their most general unifier (or the special bottom element *null formula*) and the least upper bound being their most specific generalizer (or the special top element *universal formula*).

Next we enhance the language. Our language is the following set $\mathcal{C}$ of atomary and non-atomary formulas (plus the universal and the null formula): the first order language – modulo renaming – made of standard Prolog variables, function and predicate symbols, and a unary operator -. Atoms Q of this language we call *positive contexts*, and non-atoms -Q we call *negative (negated) contexts*. The universal formula is represented here by _, and the null formula is represented by -_.

It remains to enhance the partial ordering.

Definition 5 (partial ordering of contexts). *For positive contexts X and Y we define: -X $\preceq$ _, X $\preceq$ Y if X is an instance of Y, X $\prec$ -Y if X is not unifiable with Y, -X $\preceq$ -Y if Y $\preceq$ X.*

If we assign to every atom Q from $\mathcal{C}$ the set of its ground instances, and to -Q all the other ground atoms in the Herbrand base $\mathsf{B}_{\mathcal{C}}$ for $\mathcal{C}$, we obtain an isomorphism between $(\mathcal{C}, \preceq)$ and a sub-poset of $(\mathcal{P}(\mathsf{B}_{\mathcal{C}}), \subseteq)$. Due to this isomorphism, $(\mathcal{C}, \preceq)$ is a poset and the seeming asymmetry of the definition actually is a perfect duality:

X $\prec$ -Y iff Y $\prec$ -X iff X, Y are not unifiable.

Definition 6 (context subsumes). *If CQ $\preceq$ CA, then we say: the context CA subsumes the context CQ.*

We chose the names CA and CQ to suggest *annotation context* as opposed to *query context*. The interesting case is namely when an annotation context subsumes the actual query context, meaning the annotation may fire.

Definition 7 (context inheritance). *If* $X \prec Y$*, then an annotation for* Q *within* Y *shall be inherited to* X*, i. e. it is also an annotation for* Q *within* X*.*

A self-contained example of context inheritance follows.

```
:- dynamic a/0, b/0, c/0, d/0.
:- call a => b.
:- call a within parent => c.
:- call a within - parent => d.

parent :- a.
```

```
| ?- a.
/* NOPE call a ⇏ b */
/* NOPE call a within - parent ⇏ d */

no
```

```
| ?- parent.
/* NOPE call a ⇏ b */
/* NOPE call a within parent ⇏ c */

no
```

3.3 (Generalized) Context

Assertions can be divided into predicate assertions, stating properties of a predicate, and program-point assertions, stating properties that shall hold when the code at a given point in the program is about to be executed. Traditionally, the two classes have different syntax, predicate assertions being written as declarations, and program-point assertions being scattered all over the program code (like in [7]). Our annotations are predicate assertions. However, due to contexts, as defined so far, we can capture some sets of program points. For illustrations see Sec. 6.3 and Sec. 6.7.

There is only a small step necessary to be able to capture also individual program points. For example, say we have a program-point annotation $\mathcal{A}$ placed as follows:

$$\mathsf{H} :\!- \mathsf{B}_1, .., \mathsf{B}_i, \mathcal{A}, \mathsf{B}_{i+1}, .., \mathsf{B}_n. \qquad \textit{\% k-th clause for the predicate of H}$$

then we can simulate it in Nope as

$$\mathsf{call}\ \mathsf{B}_{i+1}\ \mathsf{within}\ \mathsf{H^*pp(k,i)} \Rightarrow \mathcal{A}$$

In other words, we enhance the context specification by an optional part, program-point, linked to the simple context by an asterisk. A non-asterisk term Context is regarded as an abbreviation of Context * _ meaning “any program point within the simple context Context”. The program-point is a Datalog term representing the intended program point(s) like pp(K,J), where K is the serial number of the clause in the definition of the parent goal, and J is the serial number of the body subgoal (counting each conjunct/disjunct/implication-part). The partial ordering $\preceq$ of simple contexts shall be lifted in the obvious way to generalized contexts.

3.4 Calling premiss

For the premiss of an annotation we will also use the more evocative name *calling premiss*.

Definition 8 (calling premiss subsumes goal). *We say that a calling premiss Goal with Template within Context subsumes the call port of Q (for short: subsumes Q), if at that port Q=Goal, Template succeeds soft on Q, and also Context subsumes the parent of Q.*

More formally: Let Parent be the parent goal of Q. The following three conditions must hold. If σ is the answer substitution for Q=Goal, then σ(Q)≅Q. If θ is an answer for σ(Template), then θ(Q)≅Q. Lastly, σ(Parent) $\preceq$ σ(Context).

A logical reading of this definition (leaving out the contexts) shall be:
$\forall\ \mathcal{V}(\mathsf{Q})\colon\ \exists\ \mathcal{V}(\mathsf{Template})\backslash\mathcal{V}(\mathsf{Q})\colon\quad \mathsf{X}_1 = \mathsf{Q}_1, .., \mathsf{X}_n = \mathsf{Q}_n \vdash \mathsf{X}_1 = \mathsf{G}_1, .., \mathsf{X}_n = \mathsf{G}_n, \mathsf{Template}.$
Here $\mathcal{V}$(Term) is the set of variables in Term, and $\mathsf{X}_1, .., \mathsf{X}_n$ are new variables. $\mathsf{Q}_1, .., \mathsf{Q}_n$ are respectively the first, .., nth arguments of Q, i. e. the actual calling substitution, and similarly $\mathsf{G}_1, .., \mathsf{G}_n$ are the arguments of Goal, i. e. the calling substitution monitored by the annotation. The monitored substitution together with Template constitutes the *full*, or *normalized* monitored template. So the actual calling substitution shall imply the monitored template.

Observe that this allows local variables, namely variables that don't occur in Goal can be used to create and propagate values from the calling premiss through to the constraint.

Now we can define what does it mean to apply a call annotation.

Definition 9 (applicable). *An annotation is applicable on a goal Q if its calling premiss subsumes Q.*

Definition 10 (call annotation satisfied/violated). *An applicable call annotation is satisfied/violated for a goal Q, if its constraint succeeds/fails at the call port of Q.*

Definition 11 (correctness wrt annotations and queries). *An annotated program is said to be correct if, during the computation of arbitrary queries (from the given set of allowed queries), all the applicable annotations are satisfied.*

4 Following the execution model

In the previous section we suggested annotating the call port of a goal, in order to grasp more precisely the intended calling patterns. So what about the other three ports? Here we refer to the classic model of Prolog execution [2], abstracting a goal to a black box with four ports, known from the debugging sessions.

Following this model, we propose a language of annotations as a step towards capturing procedural aspects of Prolog. This language has the advantage of there being a workable consent among Prolog users on what the box trace model of Prolog execution means.

Definition 12 (port annotation). *Let P be a predicate of arity n. A Port annotation for P/n is a Prolog goal of the form*

$$\mathit{Event} \Rightarrow \mathit{Constraint}$$

where

Event ::= *Port Premiss*
Port ::= *call* | *exit* | *fail* | *redo*
Premiss ::= *Goal* { *with Template* } { *within Context* }

Here Goal is a Prolog goal for P/n, i. e. a Prolog term of the form $P(S_1, .., S_n)$. Context is a context. Template, Constraint are arbitrary Prolog goals.

As in [5], we deal only with *partial correctness*, because we do not claim whether goals actually succeed, fail or redo. Or even get called. Also, as the transformational semantics below reveals, our annotation language provides for accessing the values of variables on call and on exit from a query, similar to [5]. Thus, our language is also a language of *binary assertions*. The actual calling pattern is accessible on the left of ⇒, and, in case of exit and redo annotations, the exit pattern is on the right. This circumvents the need for language primitives to access the call/exit values.

Applicability of a port annotation is defined as in Def. 9. To define the meaning of port annotations, we can adopt Def. 10. But observe that a premiss shall always match the *call* port of the goal (therefore we dubbed it 'calling premiss'). In this paper we do not define the concept of a port.

$P(T_1, .., T_n) \overset{\text{CALL}}{\rightsquigarrow}$
    call_soft(FullTemplate, $P(T_1, .., T_n)$) →
        (Constraint else Warning), $P(T_1, .., T_n)$
    ; $P(T_1, .., T_n)$

$P(T_1, .., T_n) \overset{\text{EXIT}}{\rightsquigarrow}$
    call_soft(FullTemplate, $P(T_1, .., T_n)$) →
        $P(T_1, .., T_n)$, (Constraint else Warning)
    ; $P(T_1, .., T_n)$

$P(T_1, .., T_n) \overset{\text{FAIL}}{\rightsquigarrow}$
    call_soft(FullTemplate, $P(T_1, .., T_n)$) →
        ($P(T_1, .., T_n)$; (Constraint else Warning), fail)
    ; $P(T_1, .., T_n)$

$P(T_1, .., T_n) \overset{\text{REDO}}{\rightsquigarrow}$
    call_soft(FullTemplate, $P(T_1, .., T_n)$) →
        $P(T_1, .., T_n)$, (true; (Constraint else Warning), fail)
    ; $P(T_1, .., T_n)$

Fig. 2. The program transformation of Nope

But still, we do give a precise semantics of our annotations, via a program transformation. Namely, we show the essentials of our implementation of the run-time checking of annotations. The implementation bases upon the following program transformation. Every annotation for a predicate P imposes a virtual transformation, a *goal replacement*, on the user's program: *Any user's goal $P(T_1, .., T_n)$, at the program points specified by* *the annotation, be they top-level or in some clause bodies, will be computed as if replaced according to the Fig. 2. The labels on the arrows indicate the respective* Port *of the annotation.*

Note that a call annotation hence corresponds to a (global) precondition, and an exit annotation to a (global) postcondition. The usual coupling is achieved via an own (local) precondition, as illustrated in Sec. 6.1.

Lemma 1 (non-interference of port annotations). *Port annotations in Nope are not interfering with the success set: A query about the user's program Π computes the same answers and in the same sequence when posed to $\Pi+\Delta$, where Δ denotes arbitrary port annotations with terminating and pure templates and constraints.*

Annotations with a non-terminating template or constraint are clearly a bad idea. But otherwise, this lemma imposes the too strong restriction of purity upon templates and constraints. In practice, users profit from annotations with side-effects. Like [7], we guarantee that no amount of annotation tweaking will change any variables of a user's query. As for the rest of dangers, we simply count on users to be sensible in their choice of side-effects. Lemma 1 follows from the program transformation of Nope (Fig. 2), where call_soft/2 and else/2 have the following definitions:

```
call_soft(Test, Invariant) :-
  copy_term(Invariant, InvariantCopy),
  once(Test),  % Test succeeds, but does it change Invariant?
  variant(Invariant, InvariantCopy).  % Other variables may be instantiated.

else(Constraint, Warning) :-  % Constraint else Warning
  \+ Constraint -> (\+ Warning -> true; true); true.
```

Here variant/2 checks whether its arguments are renamings of each other.

Lemma 2 (compositionality of port annotations). *Port annotations in Nope are compositional: If $\mathsf{Premiss}_0$ of a given annotation $\mathcal{A}_0$ subsumes the call port of* Q, *then $\mathsf{Constraint}_0$ is going to be computed at the Port_0 of* Q, *no matter how many annotations, for what ports and in what order, were given until the call.*

This lemma follows from the program transformation of Nope.

5 Enhancing the language

A special case of constraint is false: it can never succeed, but it can nevertheless be useful, as shown in [5]: If we prove that the program, containing a false postcondition for a predicate, is correct, then we know that the matching queries can never succeed (they must fail or loop). So *non-success* can be expressed as

exit Premiss $\Rightarrow$ false

and can be exploited for integrity constraints like negative examples. To also have positive examples, we need *non-failure*, which is not expressible using pre/post-conditions alone. But, as shall be shown in Sec. 6.6, in Nope this can be expressed as ff Premiss $\Rightarrow$ false

where ff is the new *'port' of finite failure*, which can be either *derived* from the basic four ports (see Sec. 6.6), or built in.

So the subset assertions of [7] look in Nope like

ff qsort([2,1],[1,2]) $\Rightarrow$ false

Similarly to finite failure, we can derive the event of building a resolvent, the *unify port* ('bootstrapping' the annotation language).

6 Illustrations

Here we give several short examples in support of our claim that the port annotations are a versatile and an intuitively appealing means to specify general mode information, to express and support the verification of hypothetical invariants, to write customized tracers, to alert to unwanted events, and generally enhance the prototyping and self-reflective capacities of Prolog.

6.1 Mergesort and its partial specification

Our introductory example shows how annotations can help in debugging. Fig. 3 gives an implementation of the predicate merge/3. The predicate is specified to expect as inputs two sorted lists of integers, and to merge them into a sorted output list. This specification is captured in the form of two annotations.

```
:- ensure_loaded(library(nope)).  % how to activate NOPE

% Precondition: the inputs should make sense
:- call merge(X, Y, Z) => sorted(X), sorted(Y)
      else format('inputs ~w and ~w must be sorted', [X,Y]).

% Postcondition
:- exit merge(X, Y, Z) with sorted(X), sorted(Y) => sorted(Z).

merge([X|Xs], [Y|Ys], [X|Zs]) :- X>Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,X|Zs]) :- X =:= Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X<Y, merge([X|Xs], Ys, Zs).
merge([], [X|Xs], [X|Xs]).
merge(Xs, [], Xs).

sorted([]). sorted([_]). sorted([X,Y|L]) :- X =< Y, sorted([Y|L]).
```

Fig. 3. Program with annotations

The implementation of merge/3 is buggy. Usually the user discovers this by asking some queries and analyzing the answers of Prolog. However, due to the annotations, 'watching over' the specification, the analysis part can be delegated to Nope. As soon as a goal is computed which violates one of the annotations, Nope alerts us to this fact.

For example, after loading the program, we can have the following interaction.

```
| ?- merge([3,1], [], _).
/* NOPE inputs [3,1] and [] must be sorted */

yes
| ?- merge([1], [2], M).
/* NOPE exit merge([1],[2],[2,1])with sorted([1]),sorted([2]) ⇏ sorted([2,1]) */

M = [2,1] ?
```

The second message of Nope is warning us that the postcondition is violated, namely the merging algorithm produces unsorted lists (the first and the third clause for merge/3 have swapped comparisons).

An avid reader might object that we do not need the template sorted(X), sorted(Y) in the postcondition, since this is already being taken care of by the precondition. This is not quite true. Note that there can be arbitrarily many annotations for a predicate, even for the same port of it, and they will be composed via conjunction. But still, each annotation is regarded as an independent entity, in the sense of not having to take into account any other annotations. So our exit annotations actually do not assume that any call annotations have to be satisfied. For example, if we omit the template above and prove the following two annotations to hold

call merge(X,Y,Z) ⇒ sorted(X), sorted(Y)
exit merge(X,Y,Z) ⇒ sorted(Z)

with respect to the program and a closed set of top-level queries, then we know that merge/3 is only going to be called with sorted inputs, and that the output is also going to be sorted. But if the call annotation fails for a certain goal, the exit annotation is still going to be checked in Nope, requiring too much of the predicate (namely, that merge/3 is always going to deliver a sorted output, regardless of inputs).

This example was about error *monitoring* by means of annotations. At almost no extra cost we can have error *diagnosis*, by means of two generic annotations [9].

6.2 Some modes

To express that \+ does not alter its argument, we use the following mode declaration

exit \+ X with copy_term(X,X0) ⇒ variant(X,X0)

This example showed how to access the call/exit values of variables.

For our next few examples we need to discriminate between a Prolog term which already is a list (i. e., all its instances are lists), and a Prolog term that can be instantiated to a list (i. e., some of its instances are lists). Let us name the former a *complete list*, and the latter an *incomplete list*. A complete list is a Prolog list structure ·(Head,Tail) with the ultimate tail [], and an incomplete list can be either a variable or a so-called *partial list*, namely a Prolog list structure whose ultimate tail is a variable. The type predicates for complete list vs. partial list shall bear traditional names is_list/1 vs. is_partiallist/1 and shall be defined as follows.

```
is_list(X) :-
  var(X) -> fail; X=[]; X=[_|Y], is_list(Y).

is_partiallist(X) :-
  var(X) -> fail; X=[_|Y], var(Y); X=[_|Y], is_partiallist(Y).
```

Now we are able to pin down the mode of the classic predicate list/1:

```
:- exit list(X) with copy_term(X,X0) => var(X0); is_list(X0); is_partiallist(X0).
:- exit list(X) => is_list(X).

list([]).
list([_|T]) :- list(T).
```

claiming that a successful list/1 query has been called with a complete or an incomplete list, and will succeed with a complete list.

The constraint that the first argument of append/2 from [15] be a complete list can be expressed (and documented) through the following two annotations.

```
:- call append(X,Y) with var(X), \+ Y=[] => false else write(divergence).
:- call append(X,Y) with var(X), \+ \+ Y=[] => false else write(trivial_answers).

% append(-,_) :- !, fail. % reject incomplete lists  %% now catched by the ann's
append([], []).
append([L|Ls], List0) :- append(L, List1, List0), append(Ls, List1).
```

Note that, by using the default context, we do not need to actually check the list completeness, but only the var property, since each recursive call of append/2 will be checked. Note also how the constraint false serves to alert, at runtime, to unwanted events like ill-defined modes.

6.3 Use of contexts and parametricity

Due to parametricity and contexts, our annotations can be seen as macro definitions for program-point annotations. For example,

```
lr_alert(Q) :-
   functor(Q,F,N), functor(Q0,F,N),
   (call Q within Q0 => \+ subsumes_noc(Q,Q0) else bark, write(lr:Q)).

subsumes_noc(X,Y) :- groundcopy(Y,Yg), X=Yg.

:- lr_alert(append(_,_)).
```

monitors leftmost (assuming left-to-right execution) recursive calls of append/2, like

```
| ?- append(X,[Y]).
/* NOPE lr:append(_347,[_94]) */
Prolog interruption (h for help)? c
/* NOPE lr:append(_758,[_94]) */
Prolog interruption (h for help)? a
Execution aborted
```

6.4 Uniform conditioning

As a hint on the merits of conditioning even the call annotations (in order to express dependencies between the input arguments), see

```
call functor(T,F,N) with var(T) ⇒ nonvar(F), nonvar(N)
```

Observe that here we can get rid of the template, via a less readable but equivalent

```
call functor(T,F,N) ⇒ var(T) -> nonvar(F), nonvar(N); true
```

However, this does not hold in general. Namely, the *call* values of variables are normally not identical with their exit values. Thus, templates are not redundant, even in case of pure logic programs.[1] For programs with side-effects just more so, because (in case of exit, fail and redo) on the left of the arrow we have the state *before*, and on the right of the arrow: the state *after* the computation of Q.

Perhaps more convincing will be a less trivial example. The mode specification for the 'univ' operator for term creation and decomposition comprises seven error-cases in Standard Prolog (8.5.3.3.a-g in the source [8], or Sec. 98 in the reference book [4]). These can be expressed in Nope quite readably, especially since the 'natural' preconditions var(T) or Y\==[] can be separated from the constraints.

```
:- call T=..[] with var(T) ⇒ false else domain_error(non_empty_list). % case f

:- call T=..[X|Y] with var(T) ⇒ length(Y,N), Max_arity=255, N=<Max_arity else representation_error(max_arity). % case g

:- call T=..List with var(T) ⇒ \+ var(List), \+ is_partiallist(List) else instantiation_error.
% case a modified, see examples below

:- call T=..[X|Y] with var(T) ⇒ nonvar(X) else instantiation_error. % case c

:- call T=..List ⇒ var(List); is_list(List); is_partiallist(List) else type_error(list). % case b corrected as in [4]

:- call T=..[X|Y] with Y\==[] ⇒ var(X);atom(X) else type_error(atom). % case d

:- call T=..[X] ⇒ \+ compound(X) else type_error(atomic). % case e
```

The differences between this specification and the specifications of [8] and [4] can be summarized in the following table.

```
| ?- f(a) =.. L.  % [8] finds type_error(list), [4] and Nope find no error.
| ?- T =.. a.     % All find type_error(list), [4] additionally finds instantiation_error.
| ?- T =.. L.     % [8] finds type_error(list), [4] and Nope find instantiation_error.
```

[1] Another reason is the subsumption on the left vs. unification on the right of the arrow, but that is not a vital design decision of the language, the way the call/exit values are.

6.5 Tracing

The Byrd tracer (see Sec. 6.7) is a special case of annotation composition. We do not have to bear with all four ports of a general goal pattern if we only want a specific port of a specific goal pattern traced.

```
fail_alert(X) :- fail X => write('Fail: '), write(X), nl.
:- fail_alert(p(X)).

p(1) :- p(2), p(3).
p(2) :- p(4).
p(4).

| ?- p(K).
Fail: p(3)
Fail: p(4)
Fail: p(2)

K = 2 ?
```

6.6 Finite failure

The fail-port is capturing 'no (more)' rather than 'no'. Often we are interested only in genuine 'no', the finite failure. We can derive the *whole* of this new event from the basic four events, for example by emulating the Prolog stack:

```
ff(( Q with T within Ctx => C )) :-
   (call Q with T within Ctx => asserta(lemma(Q:call))),
   (redo Q with T within Ctx => asserta(lemma(Q:redo))),
   (exit Q with T within Ctx => once(retract(lemma(_:_)))),
   (fail Q with T within Ctx => once(retract(lemma(Q:What))), What=call -> C; true).
```

Alternatively, we can provide finite failure as a regular annotation, say ff, on a par with call/exit/fail/redo. This is a matter of only adding a new entry in the transformation table of Nope, and allows for superior code. Either way, we may now write

```
ff_alert(X) :- ff X => write(finitely_fails:X), nl.
:- ff_alert(p(_)).
```

and obtain the desired behaviour:

```
| ?- p(K).
finitely_fails: p(3)

K = 2 ?
```

Non-failure, which is not expressible using pre/post-conditions alone, can now be expressed as ff Premiss $\Rightarrow$ false.

In a similar way, we can derive the event of building a resolvent, the *unify port*, from the four basic events.

6.7 Byrd invariant

Correctness of the original Byrd's trace algorithm [2] depends essentially on the claim that only one flipflop/0 predicate, defined as alternatively succeeding and failing, suffices to handle arbitrarily many predicates. Of the annotation languages known to us, only [7] can express such a claim, by resorting to program-point assertions. Contexts make this expressible in a predicate assertion.

```
:- dynamic tracing/0, notracing/0.
flipflop :- retract(tracing), assert(notracing).
flipflop :- retract(notracing), assert(tracing), fail.
tracing.

% Byrd's trace invariant
:- call call(Goal) within break(Goal) => notracing. % for any Goal

break(Goal) :-
  trace(call, Goal), (call(Goal); trace(fail, Goal), fail),
  (trace(exit, Goal); trace(redo, Goal), fail).

trace(Port, Goal) :- tab(1), write(Port:Goal).

% Example. For every traced predicate a new first clause will be asserta'ed.
p(X) :- flipflop, !, break(p(X)).
p(1) :- p(2), p(3).
p(2) :- p(4).
p(4).
```

6.8 Safe negation

Here we show a special case of the safe negation problem. A non-directed graph is given as a set of edges, written as a symmetric relation edge/2. We would like to know if the following program for transitive closure is 'safe', i. e. whether it would compute only sound answers.

```
% path(U, V, P) :-
%    P represents a cycle-free path between the nodes U and V as a list [U,..,V].
%    Cycle-free means here: No node occurs in P twice.

path(U, V, P) :- % safe to use in all modes
  path1(U, [V], P).

% path1(U, [V|Visited], [Un, .., U1, V | Visited]) :-
%   [Un, .., U1, V] is a path from U to V. No Ui occurs in [V|Visited]. No Ui occurs twice.
%  Visited must be 'ground', i.e. may contain no variables.

path1(U, [V|Visited], [U, V|Visited]) :-
  edge(U, V), \+ member(U, [V|Visited]).

path1(U, [V|Visited], Path) :-
  edge(W, V), \+ member(W, [V|Visited]), path1(U, [W, V|Visited], Path).
```

Lemma 3 (path:nonfloundering). *The previous program does not flounder (wrt an arbitrary* path/3 *query), since the following annotation holds of it:*

call \+ member(X, L) ⇒ ground(X), ground(L)

This annotation is a consequence of the following two easier-to-prove ones:

exit edge(X, Y) ⇒ ground(X), ground(Y)
call path1(U, [V|Visited], W) ⇒ ground(Visited)

A straightforward inductive proof of the correctness of the program wrt the last two annotations and arbitrary path/3 queries follows from the verification condition of [5], being that we only have here a pre and a post condition. For the general case of port annotations, an extension to this verification condition can be proved.

7 Related work

Due to lack of space, we discuss in this paper only approaches with, in our view, 'strong declarative emphasis', and left out the 'more procedural' ones, which are covered to some extent in [9].

Mode languages The DEC-10 modes are de facto standard, which is reflected in the Standard Prolog sources [8]. The big asset of these modes is their simplicity, a very important criterion for a mode language, given the reluctance of users to put down any declarations at all. In [16] more precise (and complex) specification was possible. The trend of inventing new, finely tuned mode symbols was perpetuated in some other dialects of Prolog, with the result that mode declarations became rather arbitrary and complicated to use.

Somogyi [18] brought a vital simplification: the two implicit components of using a predicate, namely the calling pattern and the exit pattern, should be separated. Here, a mode is a *mapping* from an initial to a final state of instantiation of the arguments of the predicate, such that nothing gets less instantiated. This is a refinement of the classic concept of modes, making them simple and flexible. His concept of modes is scheduled for implementation within the Mercury project [6].

Modes and beyond Naish [11] initiated the research into new, general kinds of declarations. His idea was to associate with some predicates of a program Π so-called type-predicates. Intended is the following relationship: the specification Σ corresponding to Π is the same as Π except that each clause $p(T_1, .., T_n)$:– B is replaced by $p(T_1, .., T_n)$:– B, p_type($T_1, .., T_n$). He designed and implemented a concrete declarative mode language [12]. The mode language generalizes upon directional types by putting constraints on type variables.

A further major step towards the goal of more declarative rendering of Prolog code is the LPTP framework of Stärk [19], proposing a first-order language capable of stating that a general goal succeeds, fails or (universally) terminates, together with a powerful calculus (formalizing a suitable execution model) for interactively proving these properties. The inherent limitation here is purity: only a logical subset of Prolog (no cut, var or assert/retract allowed) can be treated.

A different approach was put forward by Drabent and Małuszyński [5]: they introduced a general Prolog-like language to formulate pre/post conditions for predicates of a Prolog program. Each predicate must have one pair of pre/post conditions (*assertion*), which may be arbitrarily complex. For example, a predicate p/1 not changing its argument could have the assertion consisting of the trivial precondition true, and the postcondition $.p_1 = p_1.$, meaning that upon exiting the predicate p, the corresponding calling pattern (designated by the left-dot) of the only argument of the predicate (designated by p_1) has to be identical to the exit pattern (designated by the right-dot). Here the dot syntax serves to differentiate between the two argument values, call and exit values. The authors coined the term *binary assertions* for assertions that have the means of accessing these two sets of argument values. The most popular language in the spirit of this general scheme is from the generic framework CIAO of Hermenegildo et al. [7]. Their language of assertions is a subset of Prolog. This has the distinct advantage of avoiding the introduction of an entirely new language, which would have to be learned (and liked) first. Here are some examples of their annotations.

```
:- calls qsort(A,B) : list(A).
:- success qsort(A,B) : list(A) => list(B).
:- comp qsort(A,B) : (list(A), var(B)) + does_not_fail.
```

The first annotation is a (global) precondition, the second one a postcondition corresponding to the concept of assertion in [5], since it accommodates an own precondition, given before the arrow. The third annotation serves to express some properties of the computation like non-failure. Behind the somewhat patchworky syntax is a more serious issue of expressive power: pre/post conditions alone do not suffice to express every property of execution, e. g. non-failure. As stated in [7], *"no property which refers to (a sequence of) intermediate states in the computation (...) can be easily expressed using calls and success assertions only".* This shortcoming of the pre/post conditions is here being compensated for by handing the more tricky properties over to Prolog.

Arguably a more natural way is to start from an explicit execution model of Prolog. The model should ideally be simple and complete. One such model happens to be generally known. The 4-port model [2], which is the basis of the standard Prolog debugger, is a *complete* execution model of Prolog, in the sense of entailing every aspect of Prolog execution necessary for verification, and therefore all the information one might need. This model maps a query Q to its standard trace $\mathcal{T}(Q)$, which is a sequence of *events* of the form Port $F(T_1, .., T_N)$, where Port may be one of {call, exit, fail, redo}, F/N are predicates, T_i are terms, representing the 'atomary' steps of the Prolog interpreter during the execution of Q (as defined in [2]).

This idea originates in probably the first proposal of Prolog annotations ever made, O'Keefe's public domain program Advice [14]. The annotations of Advice are of the form advise(Goal, Port, Action), which supersedes pre/post conditions.

Comparison to Nope The generalized declarations of [11] would correspond to an exit annotation in Nope. A concept like complete list (see Sec. 6.2) cannot be expressed in [12], types being closed under instantiation. Regarding [19], our annotation language cannot state e. g. that a goal succeeds, only that, *if* the goal succeeds, then a constraint must hold. On the other hand, [19] is confined to pure logic programming so cannot represent degrees of instantiation.

Now we round off the comparisons to the three most closely related languages to our own, [5], [7] and [14]. Our approach accommodates pre/post conditions in the form of binary assertions as pioneered in [5], and also uses Prolog as the annotation language like [7]. But in contrast to these, our approach is taking into account that our object language is not an arbitrary one but Prolog, with its specific procedural semantics, so we stretched the general idea of pre/post conditions to accommodate specific events from a well-known *model* of Prolog execution. Therefore we can more naturally express procedural properties of Prolog. As an additional bonus, our annotations are written in terms of procedural concepts known by everybody who ever tried the Prolog debugger. One novelty of our approach is that each one of the annotations can be restricted to a subset of calls by means of templates and contexts (*calling premiss*). All annotations are uniform and *parametric*, since they are defined as Prolog goals, which makes them particularly suitable for dynamic checking. The novel idea of *context* allows expressing program-point assertions. The annotations are applied via a general kind of matching instead of unification.

On the other hand, [7] allows the user to choose between two checking regimes for pre and post conditions, one using their variant of subsumption akin to our soft success (but without local variables), and the other using unification (doubly negated to prevent interference). Default is the subsumption. In our approach, templates are checked using subsumption ('for all') and constraints using unification ('exists').

Advice [14] differs from Nope mainly in that it has unary assertions (i. e. the calling pattern is not available), uses unification instead of matching, has no contexts, and negated occurences of a predicate cannot be checked on their own (as no built-ins can be checked). The second item is of course an advantage of Advice over Nope, since matching can be emulated by unification but not vice versa.

8 Outlook

Similar to [3] and [7], we advocate dividing the burden of analysis between static and dynamic phases. The outline of our scenario: 1) Static analysis of some acceptable classes of annotations, with respect to the cost of analysis and the realistic gain for the user. The shining example of a widely accepted static check in logic programming is the singleton variable warning [13]: a variable occuring only once in a clause indicates a probable programming error. Quick and useful, this test[2] became classic. 2) Having a stock of basic static tests, which are being taken care of by fast algorithms, the next objective is to provide some kind of calculus, enabling us to prove more complex annotations from the basic ones. 3) The rest of annotations are checked at runtime.

We envisage much further work on annotation languages for describing procedural properties of Prolog, especially in view of practical relevance. A competitive instance of Prolog has to accommodate foreign languages and programming paradigms like modules, constraints or agents, getting ever more complex on the way.

[2] amusingly enough, even this could be represented via port annotations, monitoring the predicates for loading code, e. g. call assertz(C) $\Rightarrow$ \+ thereis_singleton_in(C).

Acknowledgments

Many thanks for valuable comments upon previous drafts of this paper and other bits are due to C. Beierle, G. Meyer, R. Stärk, I. Đurđanović, M. Hermenegildo, U. Neumerkel, M. Carlsson, Vuk and the anonymous referees. This research was supported by the Deutsche Forschungsgemeinschaft (DFG), grant Be 1700/3-1.

References

1. Edinburgh University A. I. Applications Institute. Edinburgh Prolog tools. ftp://sunsite.doc.-ic.ac.uk/packages/prolog-pd-software/tools.tar.Z, 1988.
2. Lawrence Byrd. Understanding the control flow of Prolog programs. In S. A. Tärnlund, editor, *Proc. of the Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, 1980. Also as D. A. I. Research Paper No. 151.
3. P. W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *J. of Logic Programming*, pages 31–69, 1992.
4. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (reference manual)*. Springer-Verlag, 1996.
5. W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.
6. F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. *The Mercury Language Reference Manual*. http://www.cs.mu.oz.au/research/mercury, 1998. Release 0.8.
7. M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 1999.
8. ISO Prolog Standard drafts. http://www.logic-programming.org/prolog_std.html. Part I.
9. M. Kulaš. Debugging Prolog using annotations. In *WLPE'99: 10th Workshop on Logic Programming Environments*, Las Cruces, NM, 1999. Also in ENTCS, http://www.elsevier.-nl/locate/entcs.
10. Gregor Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Informatik Berichte 200, FernUniversität Hagen, 1996. See also tutorial notes of ICLP'97. Source distribution http://www.fernuni-hagen.de/pi8/typical.
11. L. Naish. Types and intended meaning. In F. Pfenning, editor, *Types in Logic Programming*, pages 189–215. MIT Press, 1992.
12. L. Naish. Mode checking using constrained regular trees. Technical Report 98/3, Department of Computer Science, University of Melbourne, 1998. Also as http://www.cs.mu.oz.au-/~lee/papers/modealg/.
13. Richard A. O'Keefe. *vcheck.pl*. In DEC-10 Prolog Library [1]. Check for misspelled variables, 1982.
14. Richard A. O'Keefe. *advice.pl*. In DEC-10 Prolog Library [1]. Interlisp-like advice package, 1984.
15. Quintus Corp., Palo Alto, CA. *Quintus Prolog Language and Library*, 1991. Release 3.1.
16. Quintus Corp., Palo Alto, CA. *Quintus Prolog Reference Pages*, 1991. Release 3.1.
17. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*. Edinburgh University Press, 1970.
18. Z. Somogyi. A system of precise modes for logic programs. In *Proc. of the ICLP*, 1987.

19. Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/-personal/staerk/lptp.html. Release 1.05.
20. D. H. D. Warren. Implementing Prolog – Compiling predicate logic programs. Research Reports 39 and 40, University of Edinburgh, Dept. of Artificial Intelligence, 1977.