Abstracting Builtins for Groundness Analysis

Andy Heaton and Andy King

School of Computer Studies, University of Leeds, LS2 9JT, UK.

Computing Laboratory, University of Kent, CT2 7NF, UK.

Abstract

This note clarifies how to handle solution gathering meta-calls, asserts and retracts in the groundness analysis of Prolog.

1 Introduction

Most work on static program analysis for Prolog has concentrated on the design of abstract domains and their operations, rather than issues of how builtins such as meta-calls and dynamic predicates should be handled. Many realistic programs contain such builtins, however, and so this is an important issue to address when constructing an analyser [2–4,6,7]. This note details how builtins are handled in a groundness analyser developed at the University of Kent in collaboration with the Universities of Ben-Gurion and Leeds. This analyser is composed to two core modules: an abstracter module which takes, as input, a program and produces, as output, an abstract version of the program that only expresses grounding dependency information; a fixpoint engine which traces the dependencies in the abstract program to infer which arguments of the input program are ground. Builtins pose (at least) four problems for static analysis and, in particular, program abstraction:

meta-call problem The problem with a goal such as call(G) is that the principal functor of the goal G might not be known until run-time and thus we cannot in general trace the call to G and deduce its answer (and those its possibly generates through sub-goals). Analogous safety problems can occur with bagof, findall, once, not and setof.

solution gathering problem A particular problem that occurs in solution gathering meta-calls such as findall(T, p(S, T), B) is that they cannot simply be abstracted as p(S, T) or even p(S, B). This is because the goal findall(T, p(S, T), B) neither instantiates S nor T (though it might ground B). Analogous problems occur with bagof and setof.

assert problem The problem with assert goals is that they can extend the program at run-time and thereby introduce new calls and answers. Specifically, suppose that a program consists of the facts p(c) and q(_) and the clause r(X) :- assert((p(Y) :- q(Y))), p(X). The query r(X) will call the assert goal. This asserts the clause p(Y) :- q(Y) which, in turn, introduces a new computation path through q(Y) (which has true as its call and answer patterns) and leads to answer pattern of true for the top-level goal r(X). If the assert was merely ignored, then call and answer patterns for q would be missed, and an incorrect answer pattern for r would be inferred.

retract problem The problem with a retract goal is basically one of precision. Consider a program that shares data between two program points using dynamic predicates to implement a blackboard mechanism. The call assert(data(I)) will write (and extend) the blackboard and elsewhere in the program the call retract(data(O)) will read (and prune) the blackboard. The goal retract(data(O)) can be safely abstracted by $true$. Precision could be improved, however, if we can infer that I is ground when the goal assert(data(I)) is called because then retract(data(O)) must ground O.

To resolve the meta-call problem (and much of the assert problem) we follow the elegant analysis model set out in [3,4] in which there are assumed to be two versions of the program: one (virgin) program that is unanalysed; and another (renamed) version that is completely analysed. The idea is for the renamed version to only express information about the calls that the analysis is able to trace. The renamed program is constructed by substituting each atom (that is not a builtin) in the virgin program, $G = p(t_1, \ldots, t_n)$, with new atom, $G' = p'(t_1, \ldots, t_n)$. Henceforth, G' will denote the rename of G. Top-level calls, as such those specified in the export declarations of a module interface, are directed at the renamed predicates. Renaming ensures that calls in one version of the program cannot normally invoke predicates in the other. One exception is with meta-calls which occur in the renamed program. These build unrenamed goals and thus call the virgin program. The other exception is with asserts that occur in the renamed program. The bodies of asserted clauses are composed of unrenamed goals and therefore can call into the virgin program. The important point is that these calls (and those they generate) do not need to be tracked to safely reason about the call and answer patterns of the renamed program. This means that, if desired, a meta-call such as call(G) can safely be ignored during analysis. Note, however, that precision may be improved by replacing call(G) with the goal G' if its principal functor is known. To summarise, the two program model of [3,4] essentially buys safety at the expense of doubling the size of the program.

With this model in mind, section 2 details how meta-calls and solution gathering goals are handled, and sections 3 and 4 explains how assert and retract goals are dealt with. As far as we aware, previous work has not considered the solution gathering problem and also the literature on handling assert and retract contains a number of holes. The appendix lists the groundness abstractions for a set of (less problematic) builtins. Section 5 concludes.

2 Meta-call and bagof problem

The two program model of [2,4] enables meta-calls to G to be (essentially) ignored during analysis. If the principal functor of G is known, however, it is usually better to replace the meta-call with a call to G' as is described below.

2.1 The call, once and not goals

Goals such as call(G) and once(G) can be handled as if they were G'. Goals such as not(G) and \+(G), however, are replaced with the goal $p(X_1, \ldots, X_n)$ where p is a new predicate symbol and $var(G) = \{X_1, \ldots, X_n\}$. The new predicate is defined as the clause $p(X_1, \ldots, X_n)$:- G', !, fail and the fact $p(X_1, \ldots, X_n)$. This essentially unfolds the definition of not(G').

2.2 The bagof, setof and findall goals

These goals are not entirely straightforward to abstract since they cannot be treated as normal meta-calls. The meta-call bagof(T, G, B) binds B to a list of instances of the template T generated through all the proofs of the goal G. (To simplify the presentation, we assume that T and B are variables.) The meta-call fails if G fails. More generally, meta-calls can take the form bagof(T, Y_1^...^Y_n^G, B) where ^ denotes existential quantification. Variables that are not quantified (and do not correspond to T) can be bound by a proof of G. The goal bagof(T, Y_1^...^Y_m^G, B) can be handled by replacing it with $p(X_1, \ldots, X_n, B)$ where $var(G) \setminus \{T, Y_1, \ldots, Y_m\} = \{X_1, \ldots, X_n\}$ and p is a new predicate defined by the clause $p(X_1, \ldots, X_n, B)$:- G', copy_term(T, B). G' cannot propagate bindings through $Y_1, \ldots, Y_m$ since these variables are not arguments of p. The setof meta-call can be treated similarly.

The meta-call findall(T, G, B) differs from bagof and setof in that it always succeeds and never binds any variables of G. Furthermore, the solution list is made up of variants of the instances of T that are generated through solving G. The meta-call is thus handled by replacing findall(T, G, B) with the goal $p(X_1, \ldots, X_n, B)$ where $var(G) \setminus \{T\} = \{X_1, \ldots, X_n\}$ and p is a new predicate defined by the clauses $p(X_1, \ldots, X_n, B)$:- G', copy_term(T, B) and $p(X_1, \ldots, X_n, B)$:- ground(B).

Observe that copy_term(T, B) is *not* described by the grounding dependency $B \leftarrow T$. If it were, the groundness of T and B after the execution of the compound goal copy_term$(T, B), T = \mathrm{a}$ would be described by $(B \leftarrow T) \wedge T = T \wedge B$, which is incorrect. To handle copy_term accurately it is necessary to extend the fixpoint engine to ground B if T is ground when the goal is encountered.

3 The assert problem

The database mutation predicates are assert, retract and abolish. The retract and abolish builtins remove clauses from the database, cannot affect safety, and thus are not as problematic as assert. With the two program model, however, asserts can be handled safely.

3.1 Exploiting dynamic declarations

One simple tactic is based on inspecting the dynamic declarations of the program since it is normally only permissible to assert clauses whose head predicate symbols are dynamic [8]. If p/n is declared dynamic, then a (nop) clause $p'(X_1, \ldots, X_n)$ can be added to the renamed program so as to ensure that the answer pattern calculated for p' is safe. The nop clause is a device that is introduced temporarily for analysis: it should not appear in the code generated for the renamed program. In this scheme, the assert goals in both the virgin and renamed program must be retained and the behaviour of assert (slightly) revised. Specifically, a call assert$((H \text{ :- } B_1, \ldots, B_n))$ must both add $H \text{ :- } B_1, \ldots, B_n$ to the virgin program and add $H' \text{ :- } B_1, \ldots, B_n$ to the renamed program. This ensures consistency between the two versions of the program. The semantics of retract also needs to be amended to keep both versions of the program consistent. Note that the body atoms of both asserted clauses are not renamed and thus only generate calls into the virgin program. The asserted clauses do not therefore compromise the safety of the call and answer patterns calculated for the renamed predicates.

3.2 Exploiting principal functor information

The dynamic declarations provide a useful safety net for handling asserts. Often, however, the principal functors of all the head and body atoms of all the asserted clauses are known at compile-time. This enables asserts to be analysed without resorting to safe (albeit imprecise) nop clauses. Specifically, consider the goal assert($(H$:- $B_1, \ldots, B_n)$) and suppose that the principal functors of $H, B_1, \ldots, B_n$ are all known (rather than, say, uninstantiated variables). Then the H' :- $B'_1, \ldots, B'_n$ clause can be added to the renamed program for the purposes of analysis. As with nop clauses, the renamed clause is merely an analysis device and should not appear in the code generated for the renamed program. Also like before, the assert goals must be retained. With the modified assert semantics, this again adds H :- $B_1, \ldots, B_n$ to the virgin program and H' :- $B_1, \ldots, B_n$ to the renamed program. The chief advantage of this tactic over just exploiting dynamic declarations is that it may improve the precision of the answer pattern of an asserted clause.

In cases where we do not have full knowledge at compile-time of all the asserted clauses, we still need to resort to the nop tactic. Specifically, if a program contains an assert with a clause that contains a body atom whose principle functor is not known at compile-time, then the nop tactic has to be applied. Similarly, if the program contains a meta-call to a goal whose principle functor is not known, then the nop tactic has to be applied. This is because, in general, unknown meta-calls and body atoms may take the form assert($p(X_1, \ldots, X_n)$) where p/n is dynamic.

3.3 Exploiting groundness information

If the program does not need to contain nop clauses (for the reasons explained above), then precision can be further improved by inferring the groundness information that describes the program state at the time at which the assert goal is encountered. Specifically, for the goal assert($(H$:- $B_1, \ldots, B_n)$), the tactic is to deduce the grounding dependencies between $\{X_1, \ldots, X_m\} = var(H \text{ :- } B_1, \ldots, B_n)$ at the program point at which the assert occurs. This is achieved by inserting the call $p(X_1, \ldots, X_m)$ into the renamed program immediately prior to the assert goal where p/m is a new predicate symbol. The p/m goal records the state of the $X_1, \ldots, X_m$ variables. To ensure that the call succeeds (without binding $X_1, \ldots, X_m$) the nop fact $p(X_1, \ldots, X_m)$ is added to the renamed program. Observe that the call and answer patterns of p/m coincide. To model the effect of the bindings on $X_1, \ldots, X_m$, the clauses H' :- $q(X_1, \ldots, X_m), B'_1, \ldots, B'_n$ and $q(X_1, \ldots, X_m)$:- $p(X_1, \ldots, X_m)$ are added to the renamed program where q/m is another new predicate symbol. The fixpoint engine is (very slightly) modified to recognise the q/m clause as assert related and specifically bar the $p(X_1, \ldots, X_m)$ body atom from contributing to the call pattern of p/m. Otherwise call and answer patterns arising within the q/m are traced normally. Hence the call pattern on p/m corresponds to the answer pattern on q/m which, in turn, propagates the bindings on $X_1, \ldots, X_m$ into the asserted clause.

4 The retract problem

As previously explained, the problem with a goal such as retract$((H$:- $B))$ is essentially one of precision: retract goals can be ignored but this loses the grounding effects of matching H :- B against the dynamic database. An analogous problem occurs with clause$((H$:- $B))$ which can be interpreted as a non-mutating read of the dynamic database. If the principal functor of H is known at compile-time to be p/n and the (static and dynamic) clauses for p/n are known to be facts, then the goals retract$(H$:- $B)$ and clause$(H$:- $B)$ can be handled as the conjunction call(H), $B =$ true. A simpler tactic is applicable for goals such as retract(H) and clause(H) where H is bound at compile-time to a fact with a principal functor p/n. Both retract and clause goals can then be replaced with the call H'. This is safe because if the dynamic database contains a matching fact, then the answer pattern for p'/n will safely approximate the effect of unifying H with the database.

5 Discussion

Most goals can be handled relatively straightforwardly in program analysis: compound goals such as G_1; G_2, G_1 −> G_2, etc can dealt with by simple program transformations; table lookup can be used for most builtins; and even catch and throw can be supported [4]. Constraints can be handled straightforwardly by re-writing to three-variable form [1]. For example, $w = x + y * z$, is written to $w = x + t, t = y * z$, where t is a fresh, temporary variable. Table lookup is then used to map three-variable forms to Boolean formulae, for example $w = x + t$ and $t = y * z$ map to $f_1 = (w \leftarrow (x \wedge t)) \wedge (x \leftarrow (w \wedge t)) \wedge (t \leftarrow (w \wedge x))$ and $f_2 = (t \leftarrow (y \wedge z))$. The grounding behaviour of the constraint $w = x+t, t = y*z$ is described by $f_1 \wedge f_2$. (Temporary variables such as t can be removed by projection, for example, t can be eliminated by $\exists t.(f_1 \wedge f_2) = (w \leftarrow (x \wedge y \wedge z)) \wedge (x \leftarrow (w \wedge y \wedge z))$. However, we choose not to do this, preferring to keep the abstracter as simple as possible. This only really affects $EPos_N$.)

The real problem in handling builtins is that the large number of cases means that it is easy to accidently introduce some imprecision. An appendix thus details how our analyser handles various builtins. It is intended to help other developers in the analysis community. Other issues that have not be discussed in this note, however, are more problematic. For example, supporting programs that are broken across several files [4,5,9] is a study within its own right. At present, our analyser has no support for modules. It also cannot handle term mutating setarg/3 goals.

References

[1] N. Baker and H. Søndergaard. Definiteness Analysis for CLP($\mathcal{R}$). *Australian Computer Science Communications*, 15(1):321–332, 1993.

[2] F. Bueno, D. Cabeza, M. García de al Banda, M. Hermenegildo, and G. Puebla. Abstract Functions for the Analysis of Builtins in the PLAI System. Technical Report CLIP1/95.0, Technical University of Madrid (UPM), Spain, 1995.

[3] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data-Flow Analysis of Prolog Programs with Extra-Logical Features. Technical Report CLIP5/95.0, Technical University of Madrid (UPM), Spain, 1995.

[4] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, pages 108–124. Springer-Verlag, 1996. LNCS 1058.

[5] M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[6] A. Cortesi and G. File. Abstract interpretation of Prolog: the treatment of built-ins. In *Proceedings of the GULP Conference on Logic Programming*, pages 87–104, 1992.

[7] S. Debray. Flow Analysis of Dynamic Logic Programs. *Journal of Logic Programming*, 7(2):149–176, 1989.

[8] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, 1996.

[9] M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–812. MIT Press, 1995.

A Groundness abstractions for builtins

In order to make the appendix containable, we give representative groundness abstractions for a range of builtins. In the sequel f_i denotes the formula $\wedge var(t_i)$ where $var(t_i)$ is the set of variables occurring in the term t_i. Note that $\wedge\emptyset = true$.

The following builtins ground all their arguments: `<`/2, `>`/2, `=<`/2, `>=`/2, `=:=`/2, `=\=`/2, abolish/1, abolish/2, absolute_file_name/2, atom/1, atom_chars/2, atomic/1, compile/1, consult/1, character_count/2, close/1, current_atom/1, current_input/1, current_module/1, current_module/2, current_op/3, current_output/1, current_stream/3, ensure_loaded/1, erase/1, float/1, flush_output/1, get/1, get/2, get0/1, get0/2, ground/1, integer/1, is/2, leash/1, line_count/2, line_position/2, load/1, name/2, nl/1, number/1, number_chars/2, numbervars/3, op/3, open/3, open/4, open_null_stream/1, peek_char/1, peek_char/2, prolog_flag/2, prolog_flag/3, prompt/2, put/1, put/2, reconsult/1, see/1, seeing/1, set_input/1, set_output/1, skip/1, skip/2, skip_line/1, source_file/1, statistics/2, stream_code/2, tab/1, tab/2, tell/1, telling/1, ttyget/1, ttyget0/1, ttyput/1, ttyskip/1, ttytab/1, use_module/2, use_module/3, version/1.

The following builtins are abstracted as $true$: `@<`/2, `@>`/2, `@=<`/2, `@>=`/2, `\==`/2, break/0, callable/1, compound/1, debug/0, debugging/0, dif/2, display/1, expand_term/2, fileerrors/0, garbage_collect/0, gc/0, help/0, listing/0, listing/1, nl/0, nodebug/0, nofileerrors/0, nogc/0, nonvar/1, nospyall/0, notrace/0, otherwise/0, phrase/2, phrase/3, print/1, read/1, repeat/0, retractall/1, subsumes_chk/2, seen/0, simple/1, skip_line/0, statistics/0, told/0, true/0, ttyflush/0, ttynl/0, var/1, version/0, write/1, writeq/1, write_canonical/1.

The following builtins are described by the bottom element of the groundness domain (usually $false$): abort/0, fail/0, false/0, halt/0, halt/1.

The final table details some (non-trivial) grounding dependencies.

t_1 `=` t_2	$f_1 \leftrightarrow f_2$	t_1 `=..` t_2	$f_1 \leftrightarrow f_2$
t_1 `==` t_2	$f_1 \leftrightarrow f_2$	C(t_1, t_2, t_3)	$t_1 = [t_2 \mid t_3]$
arg(t_1, t_2, t_3)	$f_1 \wedge (f_3 \leftarrow f_2)$	compare(t_1, t_2, t_3)	f_1
current_predicate(t_1, t_2)	f_1	format(t_1, t_2)	f_1
format(t_1, t_2, t_3)	$f_1 \wedge f_2$	functor(t_1, t_2, t_3)	$f_2 \wedge f_3$
hash_term(t_1, t_2)	f_2	hash_term(t_1, t_2, t_3, t_4)	$f_2 \wedge f_3 \wedge f_4$
instance(t_1, t_2)	f_1	keysort(t_1, t_2)	$f_1 \leftrightarrow f_2$
length(t_1, t_2)	f_2	portray_clause(t_1, t_2)	f_1
predicate_property(t_1, t_2)	f_2	print(t_1, t_2)	f_1
read(t_1, t_2)	f_1	sort(t_1, t_2)	$f_1 \leftrightarrow f_2$
source_file(t_1, t_2)	f_2	write(t_1, t_2)	f_1
write_canonical(t_1, t_2)	f_1	writeq(t_1, t_2)	f_1

For many goals, partial evaluation can be used to improve precision when the arguments of the goal are partially instantiated. For example, the unification $f(X,Y) = f(U,V)$ can be reduced to the conjunction $X = U$, $Y = V$ which, in turn, is described as $(X \leftrightarrow U) \wedge (Y \leftrightarrow V)$ rather than $(X \wedge Y) \leftrightarrow (U \wedge V)$. C/3 is handled by transforming the goal into an explicit unification.