Meta-Circular Abstract Interpretation in Prolog

Michael Codish* Harald Søndergaard†

Abstract

We give an introduction to the meta-circular approach to the abstract interpretation of logic programs. This approach is particularly useful for prototyping and for introductory classes on abstract interpretation. Using interpreters, students can immediately write, adapt, and experiment with interpreters and working dataflow analysers. We use a simple meta-circular interpreter, based on a "non-ground T_P" semantics, as a generic analysis engine. Instantiating the engine is a matter of providing an appropriate domain of approximations, together with definitions of "abstract" unification and disjunction. Small changes of the interpreter let us vary both what can be "observed" by an analyser, and how fixed point computation is done. Amongst the dataflow analyses used to exemplify this approach are a parity analysis, groundness dependency analysis, call patterns, depth-k analysis, and a "pattern" analysis to establish most specific generalisations of calls and success sets.

1 Introduction

The seminal work on abstract interpretation [25, 26] was presented in the setting of flowchart programs. By the mid 1980s, abstract interpretation had also been established as a powerful tool for the analysis of functional programs [54, 1], and in the process, the theory had been popularised. Around that time it was also realised that abstract interpretation could provide a uniform approach to a wide range of program analyses for Prolog. In fact, logic programming became a busy laboratory for abstract interpretation, when it was found that the theory offered a disciplined approach to such diverse problems as the exposure of the independence, aliasing, or freeness of variables, the patterns or types of terms, the calling modes and determinacy of predicates, and the freedom from occur-check errors in logic programs (for example [11, 28, 41, 51, 52, 58]).

To harness the wide range of analyses, a number of analysis "frameworks" were suggested. These frameworks aimed at isolating what was common to all analyses from the details that were specific to a given analysis. A framework would provide

*Department of Computer Science, Ben Gurion University of the Negev, 84105 Beer Sheva, Israel

†Department of Computer Science and Software Eng., The University of Melbourne, Vic. 3010, Australia

a definition of what abstract interpretation of logic programs was about. It would be parameterised: one would create an instance by providing an "abstract domain" together with definitions of a few (abstract) primitive operations, such as unification and variable projection. The seed for much of this work was a note [39] (now lost) circulated by Neil Jones when an informal workshop on abstract interpretation was organised by Samson Abramsky and Richard Sykes in 1985.

Some frameworks [41, 62] were presented using denotational semantics while others [9, 52] had an operational semantics flavour. The frameworks turned out to be applicable to the analysis of constraint logic programs as well, and so they became vehicles for defining a truly wide range of program analyses. Importantly, they provided the blueprints for generic Prolog analysis "engines". These were working analysis tools that could be instantiated fairly easily to solve specific analysis problems for full (constraint) logic programming languages. The engines that appeared over the next ten years include PLAI [53], GAIA [43], CHINA [3], the analyser in the CLP(R) compiler by Kelly et al. [42], and Fecht's analyser [32].

In this paper we illustrate a generic approach to the abstract interpretation of Prolog. It is based on meta-circular interpretation. The interpreters and abstract interpreters that we present are working Prolog programs. The method (and applications) that we present continue a line of work pioneered by Codish and Demoen [14, 16]. Codish and Demoen's interpreter provided a concise formalisation of an idea that was current at the time: Bottom-up evaluation could be implemented by repeatedly solving clause bodies (using the facts derived so far) and then asserting the corresponding instances of their heads until the system stabilised.

"Meta-interpreter based" abstract interpretation is easy to explain and understand. It provides the reader with a ready-to-use implementation that is immediately accessible, as opposed to the more sophisticated analysis engines that are typically faster and more general, but also much more complex. Meta-interpreters can be useful starting points for the development of debuggers, tracers, profilers, and they are convenient vehicles for language extensions or for changing aspects of semantics. We argue that they offer also a versatile platform for the specification and development of applications for abstract interpretation.

The meta-circular approach can be particularly successful in the teaching of abstract interpretation, as it encourages immediate hands-on learning, giving students an opportunity to appreciate the strong synergy between the theory and practice of program analysis. Students can learn about semantics and the principles of abstract interpretation (of logic programs) and at the same time follow the development of a platform which they will subsequently use to experiment with different logic program analyses.

In teaching meta-circular interpretation it is common to start with a "vanilla" version and then show how to extend the source language or instrument the semantics, for example to count deduction steps, or to show a trace of the evaluation. These tasks are usually quite easy. The same kind of flexibility and ease of extension carries over to the abstract interpreter. Simple changes in the interpreter can determine the semantics that is being approximated. Relatively simple changes can determine whether the analyser collects information about successful queries, calls

```
solve([]).
solve([H|Goal]) :-
    my_clause(H,Body),
    solve(Body), solve(Goal).
```

Figure 1: An Interpreter for Logic Programs

that occur at runtime, or bindings that apply at given program points. Moreover, a simple change of the "logic" component of the interpreter module can determine which kind of runtime property the analyser will investigate. And finally, the strategy used for finding fixed points can be changed, again quite easily.

Our aim is mainly practical and pedagogical. We focus on the programming aspects of the meta-circular approach and we offer many examples of program analysis. In the next section we outline the approach by giving a simple example of program analysis by meta-circular interpretation. Section 3 lays a foundation. The semantics that we use is presented together with its interpreter, and we discuss "observables" such as success sets and calls. Section 4 gives many examples of abstraction. We show how a range of different abstract domains can be implemented with little effort. In Section 5, we discuss how different strategies for fixed point iteration can be incorporated. Section 6 discusses related work and concludes. All the interpreters discussed in this paper are available via the world-wide web [21].

2 Interpretation and Abstraction

Figure 1 illustrates a Prolog program which implements an interpreter for logic programs. This is essentially the classic "vanilla" interpreter for logic programs (see for example Program 17.5 in Sterling and Shapiro [59]) except that queries (conjunctions of goals) are represented as lists of atoms. So a clause $h \leftarrow b_1, \ldots, b_n$ in a program to be interpreted is assumed to be represented by a fact of the form `my_clause(`h`,[`$b_1,\ldots,b_n$`])` and a unit clause $h \leftarrow \mathit{true}$ is represented as `my_clause(`h`,[])`. As a definition of the semantics of logic programs, the interpreter is clearly circular.

Consider now a program that implements Ackermann's function, $ack(i,j)$, using successor notation for the natural numbers ([59], Program 3.9):

```
ack(0,N,s(N)).                ack(s(M),s(N),Res) :-
ack(s(M),0,Res) :-                ack(s(M),N,Res1),
    ack(M,s(0),Res).              ack(M,Res1,Res).
```

We can represent this program as three facts, one per clause:

```
my_clause(ack(0,N,s(N)), []).
my_clause(ack(s(M),0,Res), [ack(M,s(0),Res)]).
my_clause(ack(s(M),s(N),Res), [ack(s(M),N,Res1),ack(M,Res1,Res)]).
```

To make the interpreter easier to use, we can arrange that it reads in the source program, asserting `my_clause` facts in the Prolog database along the way. In the code accompanying this paper [21] this is handled by the predicate `load_file` defined in module `input.pl`. We can then use the interpreter to answer queries:

```
?- solve([ack(s(s(0)),N,s(M))]).
M = s(s(0)),            N = 0 ;
M = s(s(s(s(0)))),      N = s(0) ;
M = s(s(s(s(s(s(0)))))), N = s(s(0))
```

and so on.

For some applications it is useful to make unification explicit in the interpreter. For example, one might want to facilitate changes to the semantics of unification, '=', so that it can stand for rational-tree unification or unification with or without the occur-check. For the purposes of dataflow analysis we shall want to approximate the behaviour of unification, so it will simplify our task if we normalise the program prior to its interpretation to make all unifications explicit. This can be done as the program is read by the predicate `load_file`. For example, the second clause for the `ack` predicate will be

```
ack(A,B,C) :-
    unify(A,s(M)), unify(B,0), unify(C,Res),
    unify(D,M), unify(E,s(0)), unify(F,Res),
    ack(D,E,F).
```

In fact, in our approach, unification will be explicit *only* in user clauses like this. In the standard interpretation, `unify` is defined by

```
unify(X,Y) :- X=Y.
```

To support this explicit reference to the unification operation, the interpreter of Figure 1 is enhanced to treat `unify` as a built-in predicate whose definition is not part of the program being interpreted. Thus we add the following clause to the interpreter:

```
solve([unify(A,B)|Goal]) :- unify(A,B), solve(Goal).
```

For a particular analysis we then need to define `unify`, to specify how it should behave for a given domain of abstractions. As an example, let us choose a domain *Parity* for parity analysis, with four elements:

`zero`	denoting	$\{0\}$
`one`	denoting	$\{1\}$
`even`	denoting	$\{n \in \mathbb{N} \mid n > 0 \text{ and } n \text{ is even}\}$
`odd`	denoting	$\{n \in \mathbb{N} \mid n > 1 \text{ and } n \text{ is odd}\}$

The elements of *Parity* are used to approximate terms: $p(\delta_1, \ldots, \delta_n)$ approximates the atom $p(t_1, \ldots, t_n)$, if t_i is a term in successor notation and $\delta_i \in \textit{Parity}$ contains the corresponding natural number. Abstract unification for *Parity* is defined

```
unify(X,Y) :-                         unify(zero,Y) :-
    nonvar(Y), Y=s(T),                    Y==0.
    unify(S,T),                       unify(X,Y) :-
    (  S=zero, X=one;                     (Y==even; Y==odd; Y==zero; Y==one),
       S=one,  X=even;                    X=Y.
       S=even, X=odd;                 unify(X,Y) :-
       S=odd,  X=even                     var(Y),
    ).                                    X=Y.
```

Figure 2: Abstract Unification for the Extended Parity Domain

in Figure 2. The definition of `unify(X,Y)` is presented using four clauses corresponding to the possible structures of the second argument Y which could be: a non-variable of the form `s(_)`, the constant `0`, one of the abstract elements (`even`, `odd`, `zero` or `one`), or a variable. The definition relies on the fact that normalisation introduces calls of the form `unify(X,term)` where `X` is a variable. The need for `nonvar` (and `var`) in the definition is an artifact of using the so-called non-ground representation for programs. With this representation, care must be taken not to instantiate meta-variables or terms accidentally. Also, we need to ensure that "abstract" symbols are chosen so that they are disjoint from "concrete" symbols.

We can now view our interpreter as a parity analyser. If we look at the set of all answers for an abstract query, we find that they describe correctly the set of all answers to a corresponding concrete query. This is due to the design of the abstract domain and the fact that the unification algorithm correctly mimics the concrete unification of natural numbers with respect to parity information. For example, if we consider the query

```
?- ack(even,one,R).
```

we obtain the answer `R = odd`. Indeed, this reflects the fact that for a positive even number n, the Prolog evaluation of the query `ack(n,1,R)` assigns to `R` an odd value, greater than 1.

However, the resulting analyser is far from satisfactory. The answer `R = odd` will be produced infinitely often and so, strictly, we cannot know whether there is also an answer of the form `R = even`, say. Moreover, a small change to the definition of `unify` such as swapping the two clauses for `unify(even,Y)` will lead to non-termination, and no answer produced at all.

Instead of considering an interpreter which embodies a top-down semantics, one alternative is to consider an interpreter which corresponds to a bottom-up semantics. Such an interpreter generates new facts incrementally, in the style of the well-known T_P operator [61] or its non-ground version, the s-semantics [29]. This type of interpreter does not require the user to specify a particular entry point, or shape of query. Rather it will produce information pertaining to all predicates in one go. This avoid the volatility of the naive top-down approach.

For Ackermann's function, the success set of the `ack` predicate is clearly infinite, so generation of all ground atomic consequences is not possible in finite time. But the approximation that we considered above will work well and it provides useful information about the success set in finite time.

The interpreters presented later are based on various variants of a T_P semantics. Presenting the Ackermann program to any of the interpreters that approximate success sets yields the following answers (though not necessarily in that order):

```
ack(zero, zero,  one)          ack(even, zero, odd)
ack(zero,  one, even)          ack(even,  one, odd)
ack(zero, even,  odd)          ack(even, even, odd)
ack(zero,  odd, even)          ack(even,  odd, odd)
ack( one, zero, even)          ack( odd, zero, odd)
ack( one,  one,  odd)          ack( odd,  one, odd)
ack( one, even, even)          ack( odd, even, odd)
ack( one,  odd,  odd)          ack( odd,  odd, odd)
```

Note that this output exposes non-trivial parity dependencies amongst the arguments of `ack`. For example, the last column on the right tells us that $ack(i, j)$ is indeed odd (and greater than 1) for all $i > 1$.

3 Concrete Semantics and Interpreters

In the previous section we sketched the use of a (standard) meta-circular interpreter for logic programs as a step in obtaining an example analyser for the language. The key point in this approach is that the interpreter plays the role of a semantic definition. By replacing the basic operations in the interpreter with operations on data descriptions, we obtain an abstract interpreter for the language. However we have seen that this approach is not satisfactory. It is similar to the common statement that, intuitively, abstract interpretation is like "executing the program over a domain of data descriptions instead of over the data itself." But the object we want to approximate is not the execution of the program as such, but rather some aspect of its meaning—namely, a semantics which makes observable the property of interest, and preferably little else. So instead of abstracting the operations in a standard meta-interpreter, we should abstract the operations in an interpreter for a suitable concrete semantics.

3.1 A Semantics Based on Fixed Points

The T_P "immediate consequences" operator [61] is the best known fixed point characterisation of the meaning of (definite) logic programs. Letting $\mathcal{H}$ denote the set of ground atomic formulas (over an appropriate alphabet), $T_P : \mathcal{P}(\mathcal{H}) \to \mathcal{P}(\mathcal{H})$ is defined by

$$T_P(I) = \left\{ A \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_k \text{ is a ground instance of a clause in } P, \\ \{B_1, \ldots, B_k\} \subseteq I \end{array} \right\}$$

Thus the least fixed point of T_P makes the program's least Herbrand model observable. In Section 2 we argued that this semantics is a better and simpler basis for abstract interpretation than the usual query-driven interpreter which requires some careful crafting to ensure termination, even for finite abstract domains. Compared to the SLD semantics [45], T_P is more abstract, hiding the resolvents that are created during SLD resolution. If needed, a T_P style semantics can be devised to make a variety of runtime events observable. Moreover, unlike an SLD characterisation, it does not force us to select an entry point in the program, or a particular query instance. Instead it generates information about all predicates.

But the T_P semantics is not without its problems. First, it makes the meaning of a program be relative to an alphabet. The meaning of the program P:

```
p(X).
q(a).
```

depends on the alphabet, which may be determined by the symbols occurring in the larger program and hence on the predicate's context. In this sense, the T_P semantics is not modular. Moreover, in the context of the non-ground meta-circular approach, an unintended interference occurs when program and interpreter are combined [50]. Let V_P denote Figure 1's interpreter together with the clauses `my_clause(p(X),[])` and `my_clause(q(a),[])`. The T_P semantics of V_P then includes `solve([p(q(a))])`, while `p(q(a))` is not in the T_P semantics of P.

Second, almost all the interesting program analyses for logic programs aim at detecting how logic variables are used or instantiated. The T_P semantics, at least in its original form, provides ground atomic formulas, and hence cannot say anything about variables. An early objection to the use of T_P as a basis for program analysis was therefore that, at least in its original form, it is of rather limited use [48, 5]. It was later shown that the introduction of a class of "non-term" constants in the Herbrand universe can ameliorate this problem, so that many interesting program properties can be captured [33]. Even so, the T_P semantics is unable to capture properties of the so-called computed answers [45], a set which, in general, is not closed under substitution.

3.2 Observing Answers

The s-semantics [7, 30, 31] is a non-ground version of the T_P semantics. The s-semantics of a program P is a set of possibly non-ground atoms which characterises both declarative as well as operational properties of a program. In particular: (a) the ground instances of the s-semantics gives precisely the least Herbrand model of P; and (b) the computed answer substitutions for any initial query q with P are determined by solving q using the (potentially infinite) set of atoms in the s-semantics of P. Letting $\mathcal{H}_v$ denote the set of atomic formulas, the s-semantics of P is defined as the least fixed point of an "immediate consequences" operator $T_P^v : \wp(\mathcal{H}_v) \rightarrow \wp(\mathcal{H}_v)$, similar to the standard least Herbrand model semantics.

More precisely,[1]

$$T_P^v(I) = \left\{ h\theta \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in P, \\ \langle a_1, \ldots, a_n \rangle \ll_C I, \\ \theta = mgu(\langle b_1, \ldots, b_n \rangle, \langle a_1, \ldots, a_n \rangle) \end{array} \right\}$$

where $\langle a_1, \ldots, a_n \rangle \ll_C I$ expresses that $a_1, \ldots, a_n$ are variants of elements of I renamed apart from C and from each other.

As a basis for designing program analyses, the s-semantics has proven particularly useful (see for example [5, 13]). This is due, in part, to the fact that it captures both declarative and operational properties, and in part to the simplicity of its definition. The advantages of using a non-ground semantics become more tangible in cases where we wish to *compute* program meanings, as in the case of semantics-based program analysis.

As an example, for the append program,

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

we get as the least fixed point:

$$\left\{ \ \mathtt{append}([x_1, \ldots, x_n], ys, [x_1, \ldots, x_n | ys]) \;\middle|\; n \geq 0 \ \right\}$$

Logically, the variables in these non-ground atoms are interpreted as being universally quantified. They obtain values from the underlying term algebra which is determined by the given alphabet. An important advantage of using a non-ground semantics is that the meaning of a program fragment is determined by the symbols it contains without considering the entire underlying alphabet. This resolves the interference problem in the sense that the meaning of a program P contains the atom p if and only if the meaning of V_P contains `solve(`$[p]$`)` [44, 50].

If we consider the instances of the atomic formulas given by the s-semantics, we get the set of "correct answers" [45], representing all the logical consequences of P. This semantics is often referred to as the c-semantics [30] and is also useful as a basis for program analysis.

The fundamental operation in the bottom-up semantics of a program P is the repeated application of facts derived so far, to solve the bodies of P's rules, generating new instances of their heads. This process starts with an empty set of facts and proceeds (sometimes forever) until no new facts are derived. Codish and Demoen [16] captured this idea with the simple meta-circular interpreter shown in Figure 3.

As for the standard interpreter of Figure 1, a clause $h \leftarrow b_1, \ldots, b_n$ in the program to be interpreted is represented as a fact `my_clause(`h`,[`$b_1, \ldots, b_n$`])`. While computing the semantics, whenever a new fact F is derived, it is maintained

[1]The formal definition involves equivalence classes of atoms (equal up to variable names). We ignore this detail in this presentation.

```
iterate :-
    operator, fail.
iterate :-
    retract(flag),
    iterate
iterate.

record(F) :- cond_assert(F).

raise_flag :-
    ( flag ->  true
    ;   assert(flag)
    ).

cond_assert(F) :-
    \+ (numbervars(F,0,_), fact(F)),
    assert(fact(F)), raise_flag.
```

(a) ***the control***

```
operator :-
    my_clause(H,B),
    prove(B),
    record(H).

prove([B|Bs]) :-
    fact(B),
    prove(Bs).
prove([unify(A,B)|Bs]) :-
    unify(A,B),
    prove(Bs).
prove([]).
```

(b) ***the logic***

```
go(File) :-
    load_file(File),
    iterate,
    showfacts.
```

```
showfacts :-
    fact(F),
    numbervars(F,0,_),
    print(F), nl,
    fail ; true.
```

(c) ***using the interpreter***

Figure 3: A Bottom-Up Prolog Interpreter

in the dynamic Prolog database as a fact of the form `fact(`F`)`. This simplifies the specification of the interpreter making it easy to implement a mechanism for "solving a clause body using the facts derived so far". In particular, the operation of renaming apart the facts used to solve a clause body is obtained "for free" from the underlying Prolog platform.

The interpreter in Figure 3 is divided conceptually into three components. On the left **(a)** is the "control" component which triggers iteration of an `operator` until no new facts are derived. When a new fact gets inserted into the Prolog database, a `flag` is raised. Iteration stops when `retract(flag)` fails in the second clause for `iterate`, that is, when no new facts were asserted in the previous iteration. On the right **(b)**, the predicate `operator` provides the "logic" and the inner loop of the algorithm which for each `my_clause(`*Head*, *Body*`)` in P proves the *Body* using facts derived so far and calls the predicate `record` which adds the *Head* to the set of facts derived so far, provided it is a new fact. In this version of the interpreter, a fact F is considered new if it is not subsumed by a fact which is already recorded (and hence implements the c-semantics). The subsumes check for F is implemented by application of the built-in predicate `numbervars(F,0,_)` which instantiates the variables in F (with fresh constant symbols) and then calling `fact(F)`. This should fail if F is to be considered new (and hence does not affect the variables in the recorded fact). The distinction between `record` and `cond_assert` is unnecessary at this point, but it will prove useful when we consider variations of the interpreter. The second clause in the definition of `prove` is for the case when unification is made explicit and should be considered as a built-in predicate by the interpreter.

Bottom-up evaluation is initiated by the query `?- iterate` which leaves the result of the evaluation in the Prolog database. In **(c)** the predicate `go` facilitates the use of the interpreter. This predicate loads the program to be interpreted, initiates iteration and finally prints the derived facts on the screen. Of course this interpreter will only work in case the semantics of the program is finite.

The interpreter in Figure 3 computes a non-ground variation of the T_P semantics of the input program. The c-semantics and the s-semantics are variations (when `cond_assert` performs a *subsumes* check or a *variant* check, respectively) [7, 29]. Both of these semantic variations are widely applied in the context of deductive databases where they are sometimes referred to as the set of *generated consequences* and *irredundant generated consequences* [46].

The practicality of using Prolog's dynamic database to store facts hinges on an efficient indexing mechanism in the underlying Prolog system. Alternatives are: (1) to maintain the set of derived facts using for example 2–3–4 trees, and provide this set as an additional argument in the interpreter; and (2) to use a tabling mechanism [17].

3.3 Observing Calls

A serious drawback of bottom-up evaluation is that it focuses only on the *success patterns* of the program (answers). Typically we are interested in how the predicates in the program are called at runtime. Another problem is that we cannot ask for just those facts that are relevant to processing a given query. It may well be that we are only interested in the analysis of the program for a specific set of initial queries, but bottom-up evaluation, as described so far, will produce the entire success set, or its approximation.

The essential idea in most bottom-up methods is to combine a top-down generation of goals with a bottom-up generation of facts. In the more common "bottom-up" approach, this is achieved through a source-to-source program transformation. The magic-sets transformation [4] and other related techniques originate as an optimisation technique in the context of deductive databases. The common principle underlying these techniques is a transformational approach in which a "magic program" $P_q^{\mathcal{M}}$ is derived from a given program P and query q. The least model of the derived program is more efficient to compute (bottom-up) and contains the information from the least model of P which is relevant for the query q. It is interesting to note that the idea behind the magic program is similar to the motivation for Jones and Mycroft's minimal function graphs [40].

```
operator :-                     prove([B|Bs]) :-
    my_clause(H,B),                 record(call(B)), fail.
    fact(call(H)),              prove([B|Bs]) :-
    prove(B),                       fact(ans(B)), prove(Bs).
    record(ans(H)).             prove([]).
```

Figure 4: An Operator for Query-Directed Bottom-Up Evaluation

Assuming that the body of a rule is evaluated from left-to-right as with Prolog's execution strategy, the magic-sets transformation is defined as follows: Let P be a logic program and q an initial query. The corresponding magic program is $P_q^{\mathcal{M}}$. For each n-ary predicate symbol p/n in P and q, $P_q^{\mathcal{M}}$ will contain two new predicate symbols: p^c/n (capturing calls to p) and p^a/n (capturing successes). In the following, if b is an atom $p(t_1, \ldots, t_n)$, we let b^c denote the atom $p^c(t_1, \ldots, t_n)$, and similarly for b^a. For each clause $h \leftarrow b_1, \ldots, b_n$ in P, $P_q^{\mathcal{M}}$ will contain n clauses of the form

$$b_i^c \leftarrow h^c, b_1^a, \ldots, b_{i-1}^a, \text{ with } i = 1 \ldots n$$

(read: "b_i is a call if h is a call and $b_1, \ldots, b_{i-1}$ are answers") together with the clause

$$h^a \leftarrow h^c, b_1^a, \ldots, b_n^a$$

(read: "h is an answer if h is a call and $b_1, \ldots, b_n$ are answers"). In addition, $P_q^{\mathcal{M}}$ will contain the "seed" fact q^c ("q is a call") corresponding to the initial query q.

The magic-sets transformation has proven useful in the context of program analysis because the facts of the form p^c in the (non-ground) least model of a transformed program $P_q^{\mathcal{M}}$ correspond to the *calls* for the predicate of p in the original program which arise in the computations of q.

We mention the magic-sets transformation only as a device to explain the interpreter in Figure 4. In fact, no program transformation takes place in our scheme—our approach is simpler. Codish [12] shows how the bottom-up semantics and its interpreter (Figure 3) can be specialised for the case when it is applied to programs generated by the magic-sets transformation, and the result is an interpreter which provides information about the answers and calls which arise in the computations (of a program and a set of initial goals).

Its operator is shown in Figure 4. The control for the interpreter is the same as that shown in Figure 3 **(a)** and the interpreter can be used as shown in Figure 3 **(c)**, except that we need to add an argument `Query` to the predicate `go` and to make the "seed" assertion `assert(fact(call(Query)))` before calling `iterate`. In the main, the interpreter is unchanged. Figure 5 shows the result of the query `ack(2,2,A)` (for readability we have avoided successor notation). The facts are listed in the order they are derived which, for this example, corresponds to the events (calls and answers) which occur in a standard top-down evaluation.

call(ack(2,2,A))	ans(ack(1,1,3))	ans(ack(2,1,5))
call(ack(2,1,A))	ans(ack(2,0,3))	call(ack(1,5,A))
call(ack(2,0,A))	call(ack(1,3,A))	call(ack(1,4,A))
call(ack(1,1,A))	call(ack(1,2,A))	call(ack(0,5,A))
call(ack(1,0,A))	call(ack(0,3,A))	ans(ack(0,5,6))
call(ack(0,1,A))	ans(ack(0,3,4))	ans(ack(1,4,6))
ans(ack(0,1,2))	ans(ack(1,2,4))	call(ack(0,6,A))
ans(ack(1,0,2))	call(ack(0,4,A))	ans(ack(0,6,7))
call(ack(0,2,A))	ans(ack(0,4,5))	ans(ack(1,5,7))
ans(ack(0,2,3))	ans(ack(1,3,5))	ans(ack(2,2,7))

Figure 5: Calls and Answers in the Evaluation of an Ackermann Query

```
operator :-
    my_clause(Head,Body),
    fact(call(_,_,Head)),
    prove(Body),
    record(ans(Head)).

prove([pp(I,J,B)|Bs]) :-
    record(call(I,J,B)), fail.
prove([pp(I,J,B)|Bs]) :-
    fact(ans(B)), prove(Bs).
prove([]).
```

Figure 6: Query-Directed Bottom-Up Evaluation with Program Points

3.4 Observing Program Points

It is often the case that analysis is required to characterise the state of the computation at a given program point. This type of information is more refined than that obtained by characterising the calls to the predicates in the program because the same predicate can be called in different ways from different points in the program.

To support analyses which focus on program points we can alter the procedure which inputs the source program and constructs its internal `my_clause` representation. The fact that q is the j^{th} atom in the i^{th} clause is represented by an atom of the form pp(i,j,q). For example, the following program to rotate the elements of a list ([59], page 312):

```
rotate(Xs,Ys) :-
    append(As,Bs,Xs),
    append(Bs,As,Ys).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

is represented internally as:

```
my_clause(rotate(Xs,Ys),[pp(1,1,append(As,Bs,Xs)),pp(1,2,append(Bs,As,Ys))]).
my_clause(append([],Ys,Ys),[]).
my_clause(append([X|Xs],Ys,[X|Zs]),[pp(3,1,append(Xs,Ys,Zs))]).
```

The changes to the interpreter are then minimal and shown in Figure 6. Similar solutions are applied in other analysers [34, 17].

With the seed assertion `assert(fact(call(0,0,rotate([a,b,c],Ys))))` for an initial query, the result of the interpretation contains the following information for

```
unify(X,Y) :-
   var(Y), X=Y.
unify(X,Y) :-
   nonvar(Y),
   ( (Y == true ; Y == false) ->
       X=Y
   ;   (term_vars(Y,Vars), iff(X,Vars))
   ).
```

```
iff(true,[]).
iff(true,[true|Vars]) :-
    iff(true,Vars).
iff(false,[false|_]).
iff(false,[_|Vars]) :-
    iff(false,Vars).
```

Figure 7: Abstract Unification for Groundness Dependency Analysis

the three calls to append in this program at points $(1,1)$, $(1,2)$ and $(3,1)$:

```
call(1,1,append(A,B,[a,b,c]))
call(1,2,append([a,b,c],[],A))
call(1,2,append([b,c],[a],A))
call(1,2,append([c],[a,b],A))
call(1,2,append([],[a,b,c],A))
call(3,1,append(A,B,[b,c]))
call(3,1,append(A,B,[c]))
call(3,1,append([b,c],[],A))
call(3,1,append(A,B,[]))
call(3,1,append([c],[],A))
call(3,1,append([c],[a],A))
call(3,1,append([],[],A))
call(3,1,append([],[a],A))
call(3,1,append([],[a,b],A))
```

4 Meta-Circular Abstract Interpretation

In the previous sections we illustrated how interpreters for logic programs can be enhanced to make unification explicit. We showed how this approach facilitates the design of program analyses. In order to obtain a dataflow analyser for parity, all we needed to do was replace the definition of the predicate unify.

4.1 Groundness Analysis

We now show how the same approach yields a groundness dependency analyser. The new definition of unify is shown in Figure 7. The atom term_vars(T,V) expresses that V is the set of variables in the term T. This predicate is a variant of Sicstus Prolog's term_variables, adapted to the specific application (see [21]). The atom iff$(X,[Y_1,\ldots,Y_n])$ captures the Boolean function $X \leftrightarrow (Y_1 \wedge \cdots \wedge Y_n)$ (read: "X is ground if and only if $Y_1,\ldots,Y_n$ are").

This analysis is very precise, effectively representing groundness dependencies as Boolean functions. It is pleasing to see how easily such a sophisticated program analysis can be implemented using meta-circular interpretation. Abstract unification is not hard to capture, and all the other components of the interpreter remain unchanged.

Consider again the append predicate depicted as Figure 8**(a)**. The groundness analyser yields the two facts given as Figure 8**(b)**. That is, for any element append$(A,B,C) \in \mathcal{H}_v$ of the program's success set, C is ground (true) if and only if A and B are. Now consider the relation append3 (to concatenate three lists)

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

(a) The append relation

```
append(true,A,A)
append(false,A,false)
```

(b) Groundness result for append

```
append3(A,B,C,D) :-
    append(B,C,Temp),
    append(A,Temp,D).
```

(c) Appending three lists

```
call(append3(true,true,true,_))
call(append(true,true,_))
ans(append3(true,true,true,true))
ans(append(true,true,true))
```

(d) Calls and answers for append3

```
bubble_sort(Xs,Ys) :-
    append(As,[X,Y|Bs],Xs),
    greater_than(X,Y),
    append(As,[Y,X|Bs],Xs1),
    bubble_sort(Xs1,Ys).
bubble_sort(Xs,Xs) :-
    sorted(Xs).
```

(e) Bubble sort

```
call(1,1,append(_,_,true))
call(1,3,append(true,true,_))
call(4,1,append(true,true,true))
call(4,1,append(false,true,true))
call(4,1,append(true,true,false))
call(4,1,append(true,false,true))
call(4,1,append(false,false,true))
```

(f) Calls to append for bubble_sort

Figure 8: Examples for Groundness Analysis

defined by the clauses in Figure 8**(c)** and consider a query-directed analysis which specifies that the call to `append3` is of the form `append3(true,true,true,_)`, indicating that the first three arguments are ground inputs. A query-directed analysis using the operator defined in Figure 4 indicates that during a computation of this query, all calls to `append` have the first two arguments ground. Notice that there are three places in the program in which there are calls to `append` (two in the definition of `append3` and one in the definition of `append` itself). The result of the analysis is given as Figure 8**(d)**.

A similar example is the bubble sort program given as Figure 8**(e)**. The two calls to `append` in this program serve different purposes. The first breaks up a list of the form

$$Xs = [\ \underbrace{\ldots}_{As},\ X, Y,\ \underbrace{\ldots}_{Bs}\ ]$$

to find two elements X and Y that need to be interchanged; the second concatenates the two lists As and $[Y, X|Bs]$ to get back the original list with the positions of X and Y interchanged. If no such X/Y pair is found then the list is sorted.

Performing the query-directed analysis with the query `bubble_sort(true,Ys)` (indicating that the first argument is ground) does not give a very useful result, because no distinction is made between the two calls to `append` (or the third call, in the definition of `append` itself). This problem can be remedied by tracking which calls are made at which program points. Assuming that the clauses for `append` occur after the clauses for `bubble_sort` (as clauses `3` and `4`), the result of

```
record(K,F) :-
    depth(K,F,NewF),
    cond_assert(NewF).

depth_list([],[],_K).
depth_list([T|Ts],[NT|NTs],K) :-
    depth(K,T,NT),
    depth_list(Ts,NTs,K).
```

```
depth(_K,Term,Term) :-
    var(Term), !.
depth(0,_Term,_NewTerm).
depth(K,Term,NewTerm) :-
    K>0,
    Term =.. [F|Args],
    K1 is K-1,
    depth_list(Args,NewArgs,K1),
    NewTerm =.. [F|NewArgs].
```

Figure 9: Recording Facts in Depth-k Analysis

a call pattern analysis that uses the program point interpreter is given as Figure 8**(f)** (only the results for `append` are shown). Notice that in the recursive call `append(Xs,Ys,Zs)` at point $(4,1)$, either the first two arguments are ground or the third is (corresponding to $(Xs \wedge Ys) \vee Zs$) while in the two calls of the first clause, at points $(1,1)$ and $(1,3)$, the third and first two arguments are ground respectively.

4.2 Depth-k Analysis

For some well-known analyses, abstract unification is simply unification, and there is no need to come up with a new definition. We now derive two such analyses, depth-k analysis in this section, and pattern analysis in the next. In both cases, we shall need to redefine how facts are recorded.

For depth-k analysis, the idea is that terms are allowed only a certain maximal depth [57, 49]. The depth of a variable or a constant is 1, while otherwise, the depth of a term t can be defined recursively:

$$depth(t) = 1 + max\ \{depth(t') \mid t' \text{ is a proper subterm of } t\}.$$

A term with depth greater than k can be approximated by replacing subterms at that depth by fresh variables. For example, `g(f(f(V)),f(V))` is depth-3 approximated by `g(f(U),f(V))`. (This abstraction in fact falls outside classical abstract interpretation [25], as there is no *best* approximation in general. For example, `g(f(f(V)),f(V))` could equally well be approximated by `g(f(U),U)`, an incomparable depth-3 term. But the approximation is safe and the analysis is useful.)

Figure 9 shows how `record` is redefined to prune the depth of the terms in a fact before adding the fact to the the set of facts derived so far. Observe that the call is changed to `record(K,F)`, so as to pass the desired depth to the analyser and that the approximation is performed by calling `depth(K,T,NewTerm)` which takes the desired depth K and a term T and produces the pruned version $NewTerm$.

Consider the following program for the Towers of Hanoi problem ([59], Program 3.31). `A-B` indicates that we should move a disk from peg `A` to peg `B`.

```
/*  hanoi(N,A,B,C,Moves) :-
        Moves is a sequence of moves for solving the Towers of
        Hanoi problem with N disks and three pegs, A, B and C.
*/
    hanoi(s(0),A,B,C,[A-B]).
    hanoi(s(N),A,B,C,Moves) :-
        hanoi(N,A,C,B,Ms1),
        hanoi(N,C,B,A,Ms2),
        append(Ms1,[A-B|Ms2],Moves).
```

Here is the result of the analysis using a depth-10 abstraction:

```
hanoi(s(0),A,B,C,[A-B])
hanoi(s(s(0)),A,B,C,[A-C,A-B,C-B])
hanoi(s(s(s(0))),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B])
hanoi(s(s(s(s(0)))),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
hanoi(s(s(s(s(s(0))))),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,_|_])
hanoi(s(s(s(s(s(s(0)...),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
hanoi(s(s(s(s(s(s(s(0)...),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,_|_])
hanoi(s(s(s(s(s(s(s(s(0)...),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
hanoi(s(s(s(s(s(s(s(s(_)...),A,B,C,[A-B,A-C,B-C,A-B,C-A,C-B,A-B,A-C,_|_])
hanoi(s(s(s(s(s(s(s(s(_)...),A,B,C,[A-C,A-B,C-B,A-C,B-A,B-C,A-C,A-B,_|_])
```

Notice that the list of moves to solve the problem (the fifth argument) is precise for 3 disks or less. After that we get only the first part of the solution. It is apparent that for an odd number of disks the solution always starts with the move `A-B` and for an even number of disks, with the moves `A-C,A-B,C-B` (though this is not a proof). Notice that so long as the number of disks is not pruned (and hence we can determine if it is odd or even) there is always a single solution for that number of disks in the fifth argument.

By analysing the program with a combined domain that abstracts numbers with using the parity domain and lists using a depth-5 analysis we obtain a *proof* of an interesting property of the Towers of Hanoi problem. Namely, for an odd number of disks, the solution always starts with the move `A-B` and that for an even number of disks, with the moves `A-C,A-B,C-B`. With this domain the analysis gives:

```
hanoi(odd,A,B,C,[A-B])                   hanoi(even,A,B,C,[A-C,A-B,C-B])
hanoi(odd,A,B,C,[A-B,A-C,B-C,_|_])       hanoi(even,A,B,C,[A-C,A-B,C-B,_|_])
```

4.3 Pattern Analysis

This analysis maintains a single concrete atom as the approximation for each predicate in the program. For a given predicate, any pair of abstract atoms is replaced by their most specific generalisation (using anti-unification). The analysis can be viewed as a generalisation of constant propagation analysis, which uses a flat domain of constants, together with bottom and top elements. With constant propagation, if a variable may assume two different values c_1 or c_2 then it is approximated by *top*. In the pattern analysis, we have terms, or partial structures, instead of

```
record(F) :-              keep_msg(F) :-
    keep_msg(F).              \+ (numbervars(F,0,_), fact(F)),
                              functor(F,N,A), functor(F1,N,A),
                              ( retract(fact(F1)) ->
                                  term_subsumer(F1,F,MSG)
                              ;   MSG = F
                              ),
                              assert(fact(MSG)),
                              raise_flag.
```

Figure 10: Recording Facts in Pattern Analysis

constants. Approximation maintains the information common to the terms approximated, that is, it provides their most specific generalisation. Approximation of the values that a variable may assume can be relaxed repeatedly, each time yielding a more general term. The number of times a value can be relaxed is finite, but cannot be bounded a priori. The abstract domain used here is therefore an example of a domain which has infinite chains, but no infinite ascending chain.

Figure 10 shows how `record` is redefined to yield a pattern analysis. The implementation of `keep_msg(F)` covers three cases:

1. There is already a pattern at least as general as `F`. In this case the call to `\+ (numbervars(F,0,_), fact(F))` fails.
2. There is no previous pattern for the predicate of `F`. In this case the condition `retract(F1)` fails and `fact(MSG)` with `MSG=F` is asserted.
3. There is a pattern `F1` for the predicate of `F` (not as general as `F`). In this case `fact(MSG)`, with `MSG` the most specific term that generalises both `F1` and `F`, replaces `fact(F1)` in the database. It is computed by the call to the Sicstus term library predicate `term_subsumer(F1,F,MSG)`.

For the goal dependent analyser, we need two specialised versions of `keep_msg(F)` (`keep_msg(call(F)` and `keep_msg(ans(F))`) to maintain most specific patterns for calls and answers respectively.

For example, consider this program which reverses a list ([59], Program 3.16):

```
reverse([],[]).                   append([],Ys,Ys).
reverse([X|Xs],Zs) :-             append([X|Xs],Ys,[X|Zs]) :-
    reverse(Xs,Ys),                   append(Xs,Ys,Zs).
    append(Ys,[X],Zs).
```

Goal dependent pattern analysis with the initial query `reverse(Xs,Ys)` yields:

```
call(reverse(A,B))                ans(reverse(A,B))
call(append(A,[B],C))             ans(append(A,[B],[C|D]))
```

indicating that for any initial query to `reverse`, in all subsequent calls to `append`, the second argument is a singleton list.

One application of pattern analysis is so-called *structure untupling* [20]. This transformation can improve the precision of other program analyses by making invariant data-structures explicit. For an example, consider the following Prolog program ([59], Program 15.3) which reverses a list efficiently, using difference lists.

```
reverse(Xs,Ys) :-                    reverse_dl([X|Xs],Ys-Zs) :-
    reverse_dl(Xs,Ys-[]).                reverse_dl(Xs,Ys-[X|Zs]).
                                     reverse_dl([],Xs-Xs).
```

Here `Xs` will be ground if and only if `Ys` is ground, for any successful query `reverse(Xs,Ys)`. However, a groundness dependency analysis can not be expected to pick up this dependency. To do that, it would be necessary to infer the groundness dependency $(\mathtt{A} \land \mathtt{C}) \leftrightarrow \mathtt{B}$ for `reverse_dl(A,B-C)`. But groundness analysis typically considers dependencies amongst predicate arguments, not between substructures, so the desired information is not produced.

Pattern analysis for this program shows that all answers for `reverse_dl` are in fact of the form `reverse_dl(A,B-C)`. Hence the program can be specialised automatically to obtain

```
reverse(Xs,Ys) :-                    reverse([X|Xs],Ys,Zs) :-
    reverse(Xs,Ys,[]).                   reverse(Xs,Ys,[X|Zs]).
                                     reverse([],Xs,Xs).
```

Groundness analysis now gives the desired accurate result, for the original program.

5 Evaluation Strategies

Until now we have presented interpreters which calculate fixed points under a naive evaluation strategy. More efficient evaluation strategies exist and are well understood. These include semi-naive evaluation [55, 60], eager evaluation [12, 63], and techniques which take advantage of the structure of strongly connected components in the program's call graph.

For the presentation of the various abstractions, the naive approach is adequate, but it is pleasing to note that variations of the evaluation strategy are easily introduced to the implementation of an analysis by adapting the underlying interpreter.

5.1 Semi-Naive Evaluation

Semi-naive evaluation is based on the observation that whenever a clause body is solved, in some iteration, at least one new fact, derived "one iteration ago", should be used. Otherwise, the resulting instance of the clause head will not be new. The classic algorithm for semi-naive evaluation (for example as presented by Ullman [60]) maintains the facts derived so far in two sets denoted Δ_{new} and Δ_{old}. To facilitate the Prolog implementation, a new fact F derived in the n^{th} iteration is maintained in the dynamic Prolog database together with a "time-stamp" in the form `fact(n,F)`, where `n` is an iteration number.

```
iterate(Prev,Curr) :-
    operator(Prev,Curr),
    fail.
iterate(_,Curr) :-
    fact(Curr,_), !,
    Next is Curr + 1,
    iterate(Curr,Next).
iterate(_,_).

record(N,F) :-
    cond_assert(N,F).

cond_assert(N,F) :-
    \+ (numbervars(F,0,_), fact(_,F)),
    assert(fact(N,F)).
```

```
operator(_,1) :-
    my_clause(H,[]),
    record(1,H).
operator(Prev,Curr) :-
    Curr > 1,
    my_clause(H,B),
    fact(Prev,Atom),
    select(Atom,B,Rest),
    prove(Rest),
    record(Curr,H).

prove([B|Bs]) :-
    fact(_,B),
    prove(Bs).
prove([]).
```

Figure 11: A Prolog Interpreter for Semi-Naive Evaluation

Figure 11 illustrates a Prolog interpreter for semi-naive bottom-up evaluation. The interpreter is essentially the same as that presented in Figure 3. The important difference is that the `operator` predicate on the right carries two arguments, corresponding to the previous and current iteration numbers, and always selects from a clause body a fact derived in the previous iteration, before proving the rest of the body. For semi-naive iteration, evaluation is initiated by the query `iterate(0,1)` which leaves the result of the evaluation in the Prolog database.

Note that (a) to determine whether another iteration is required we check if there is a fact of the form `fact(Curr,_)`, instead of using a flag; (b) when recording a new fact we time stamp it using the additional argument in the predicate `record`; and (c) the definition of the `operator` predicate contains two clauses — one for the first iteration, and one for the subsequent iterations.

5.2 Eager Evaluation

Eager evaluation also aims to insure that at least one "new" atom is used when a clause body is solved. The difference between this strategy and the semi-naive strategy is in the book-keeping of "what is new". Eager evaluation uses a newly derived fact F immediately, postponing other tasks while exploring which new facts can be derived now that F has been established. In the process, new facts may be derived, and their investigation will then get higher priority. That is, tasks are handled in a first-in last-out manner.

An interpreter for this method was introduced by Wunderwald [63]. In contrast to semi-naive evaluation, eager evaluation does not need to distinguish between new and old facts. Instead, whenever a new instance of a clause head is derived it is used immediately to re-solve all clauses which contain a call to the corresponding predicate in their body. Consequently eager evaluation is simple to describe as a

```
go(File) :-
    load_file(File),
    eager(true),
    fail.

prove([B|Bs]) :-
    fact(B),
    prove(Bs).
prove([]).
```

```
eager(true) :-
    my_clause(H,[]),
    eager(H).
eager(Atom) :-
    record(Atom),
    my_clause(H,B),
    select(Atom,B,Rest),
    prove(Rest),
    eager(H).
```

Figure 12: A Prolog Interpreter for Eager Evaluation

recursive depth first strategy. Figure 12 depicts an interpreter for eager bottom-up evaluation. Evaluation is triggered by a call of the form `eager(true)` and the control component is managed by the underlying control for recursive calls, the runtime stack.

Eager (and semi-naive) evaluation can be used also for query directed bottom-up evaluation as shown in Figure 13. Program lines have been numbered to facilitate an explanation. There are two cases to consider when a new fact is found:

1. If the new fact is an *answer* (line `14`), then for each clause `Head :- Body` (line `15`) that has been called (line `16`) and contains a matching call for the new fact (line `17`), we solve the `Body` (line `18`) and record the corresponding instance of the `Head` (line `19`). If it is new then there is a recursive call (line `20`) (otherwise we fail at line `19`).

2. If the new fact is a *call* (line `21`), then for each clause with a matching head (line `22`), we solve the body (line `23`) and record the corresponding instance of the `Head` (line `24`). If it is new then there is a recursive call (line `25`) (otherwise we fail at line `24`).

In both cases, solving a clause body involves recording and processing new calls (lines `7-9`) and matching body atoms with with facts derived so far (lines `10-12`).

We can improve the code in Figure 13 for the case when a new answer is encountered (lines `14-20`). A new answer `Atom` that matches a call b_i in the clause $h \leftarrow b_1, \ldots, b_i, \ldots, b_n$ may give rise to new call patterns for $b_{i+1}, \ldots, b_n$ but will not influence calls for $b_1, \ldots, b_i$. Hence the case covered by lines `7-9` in the code is redundant for atoms to the left of (and including) b_i. The optimised interpreter is obtained by replacing lines `17` and `18` by

```
/*17'*/   append(Left,[Atom|Right],Body),
/*18'*/   prove(Left), prove_ind(Right),
```

respectively and defining `prove` by

```
prove([B|Bs]) :-
    fact(ans(B)), prove(Bs).
prove([]).
```

```
/* 1*/  go(File,Goal) :-
/* 2*/     load_file(File),
/* 3*/     record(call(Goal)),
/* 4*/     eager(call(Goal)),
/* 5*/     fail.
/* 6*/  go(_,_) :- showfacts.

/* 7*/  prove_ind([B|_]) :-
/* 8*/     record(call(B)),
/* 9*/     eager(call(B)).
/*10*/  prove_ind([B|Bs]) :-
/*11*/     fact(ans(B)),
/*12*/     prove_ind(Bs).
/*13*/  prove_ind([]).

/*14*/  eager(ans(Atom)) :-
/*15*/     my_clause(Head,Body),
/*16*/     fact(call(Head)),
/*17*/     \+\+ member(Atom,Body),
/*18*/     prove_ind(Body),
/*19*/     record(ans(Head)),
/*20*/     eager(ans(Head)).

/*21*/  eager(call(Atom)) :-
/*22*/     my_clause(Atom,Body),
/*23*/     prove_ind(Body),
/*24*/     record(ans(Atom)),
/*25*/     eager(ans(Atom)).
```

Figure 13: Induced Eager Evaluation

6 Discussion

We have presented the case for the meta-circular approach to the analysis of logic programs. We have argued that, when taking this approach, it is natural to use interpreters based on T_P-style semantics, or more precisely, on the s-semantics [29]. This paper presents working Prolog interpreters for program analysis, and detailed code is available via the world-wide web [21]. We have avoided details about how to approximate Prolog builtins, as this is well understood.

The meta-circular approach is highly versatile, in the sense that simple changes to the interpreter allow us to change what is being observed, which properties are being extracted, and even the evaluation strategy used. It appears particularly useful for prototyping, but also for classes on semantics and abstract interpretation of logic programs. The versatility of the approach has again been confirmed to us during the writing of this survey: Both the domain *Parity* from Section 2 and its combination with the depth-k domain used in Section 4.2 resulted from interactive experiments with the interpreter—they were created on the laboratory bench, so to speak.

A bottom-up formulation of abstract interpretation for logic programs was originally suggested by Marriott and Søndergaard [47, 48, 49]. The idea was developed by many researchers, including Barbuti, Giacobazzi and Levi [5], Codish and Demoen [14, 15, 16], Codish, Dams and Yardeni [13], Gallagher and coworkers [35, 33], as well as in the "pre-interpretation" approach [8, 10]. An interesting approach has been the use of Toupie [24], which can be thought of as a finite-domain μ-calculus interpreter allowing fixed point problems to be stated succinctly and solved efficiently. Toupie incorporates a finite-domain constraint solver and was designed and used for the abstract interpretation of logic programs.

In addition to the domains used as examples in this paper, several others can be utilised with the framework. Recent work considers E-unification (unification with an equality theory E [2]) for `unify(A,B)`. It turns out that ACI unification leads to a notion of type dependency analysis [18]; adding a unit element **1** and using standard ACI**1** unification gives an analysis for groundness dependencies over the domain of *definite* Boolean functions [36]; while a non-standard ACI**1** unification algorithm gives set-sharing analysis [19].

Benoy and King [6] apply Sicstus Prolog's CLP(R) library to infer argument size relations within a meta-interpreter similar to those we have shown here. They show how `unify(A,B)` is implemented using constraint conjunction, facts are added to the database using a convex hull operation, and projection is implemented using Sicstus Prolog's `dump/3` predicate. Codish and Taboch [23]) introduce a meta-interpreter which makes loops observable and abstracting this interpreter (for groundness and size relations information) leads to a termination analyser for Prolog (see `http://www.cs.bgu.ac.il/~mcodish/TerminWeb/`).

Building on work by Dawson, Ramakrishnan and Warren [27], Codish, Demoen and Sagonas [17] demonstrated the utility of a logic program system with *tabulation* for the purpose of program analysis. To this end they use the XSB system [56]. XSB offers efficient fixed point evaluation through tabling implemented at the abstract machine level. The authors describe simple XSB interpreters which make answers and calls observable and give several abstract domains for analysis. The approach is similar to that presented here, but the emphasis is on techniques that make the analyses efficient, such as the tabling of abstract operations. In contrast we have focused on the specification of extensions *within* the interpreters. The evaluation strategy cannot be varied in the XSB approach, as it is fixed by the underlying implementation.

A shortcoming of a dataflow analysis approach based on meta-circular interpretation is that an interpreter inevitably incurs a cost which is felt as input programs grow. However, it is possible to apply partial evaluation to remove the level of interpretation [44]. This leads to a more efficient analyser and can be viewed as what is sometimes called abstract compilation [37].

Another shortcoming is that it is far from clear how to implement abstract domains for which abstract conjunction is not simply conjunction. Set sharing analysis, for example, falls in this class. The problem is not so much the fact that the analysis needs to keep track of relations that hold amongst variables, as opposed to properties of individual variables. The same can be said for the groundness dependency analysis given in Section 4.1, and it is possible to express sharing information much like groundness information, using propositional logic [22]. Rather, the problem is that sharing analysis uses an abstract conjunction operation which is rather complex, and not faithfully mimicked by the conjunction implicitly used by the interpreter.

However, as experimental or pedagogical tool, the meta-circular approach has considerable advantages, as we hope to have demonstrated in this paper. In addition, Howe and King [38] demonstrate that it can produce extremely fast analysers. We are not aware of similar work for other programming language paradigms, and this is something that may be worth exploring.

Acknowledgements

We would like to thank John Gallagher for the fruitful discussions we have had, and Samir Genaim for help with the interpreters.

References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–126. Oxford University Press, 1994.

[3] R. Bagnara. China: A data-flow analyzer for CLP languages. Survey at URL http://www.cs.unipr.it/China/, 19 February 2001.

[4] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. Fifth ACM SIGMOD/SIGACT Symp. Principles of Database Systems*, pages 1–15. ACM Press, 1986.

[5] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based abstract interpretation. *ACM Transactions on Programming Languages*, 15(1):133–181, 1993.

[6] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J. Gallagher, editor, *Logic Program Synthesis and Transformation: Proc. LOPSTR 96*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, 1997.

[7] A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19&20:149–197, 1994.

[8] D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting s-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming: Proc. Sixth Int. Symp.*, volume 844 of *Lecture Notes in Computer Science*, pages 432–446. Springer-Verlag, 1994.

[9] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

[10] M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A freeness and sharing analysis of logic programs based on a pre-interpretation. In R. Cousot and D. A. Schmidt, editors, *Static Analysis: Proc. Third Int. Symp.*, volume 1145 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 1996.

[11] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimisation of Prolog programs. In *Proc. 1987 Symp. Logic Programming*, pages 192–204. IEEE Comp. Soc., 1987.

[12] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[13] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–125, 1994.

[14] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Logic Programming: Proc. 1993 Int. Symp.*, pages 114–129. MIT Press, 1993.

[15] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In P. Van Hentenryck, editor, *Static Analysis: Proc. First Int. Symp.*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.

[16] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. *Journal of Logic Programming*, 25(3):249–274, 1995.

[17] M. Codish, B. Demoen, and K. Sagonas. Semantics-based program analysis for logic-based languages using XSB. *Springer Int. Journal of Software Tools for Technology Transfer*, 2(1):29–45, 1998.

[18] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.

[19] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. *Journal of Logic Programming*, 41(2):110–149, 2000.

[20] M. Codish, K. Marriott, and C. Taboch. Improving program analyses by structure untupling. *Journal of Logic Programming*, 43(3):251–263, 2000.

[21] M. Codish and H. Søndergaard. Meta-circular abstract interpretation in Prolog. http://www.cs.bgu.ac.il/~mcodish/Tutorial/, 11 June 2001.

[22] M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[23] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[24] M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 75–91. Springer-Verlag, 1993.

[25] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[26] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[27] S. Dawson, R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems: A case study. In *Proc. ACM SIGPLAN 96 Conf. Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.

[28] S. K. Debray and D. S. Warren. Automatic mode inference for Prolog programs. In *Proc. 1986 Symp. Logic Programming*, pages 78–88. IEEE Comp. Soc., 1986.

[29] M. Falaschi, G. Levi, M. Gabbrielli, and C. Palamidessi. A new declarative semantics for logic languages. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 993–1005. MIT Press, 1988.

[30] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[31] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 103:86–113, 1993.

[32] C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[33] J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. Lloyd, editor, *Logic Programming: Proc. 1995 Int. Symp.*, pages 351–365. MIT Press, 1995.

[34] J. Gallagher and D. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation: Proc. LOPSTR 92*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.

[35] J. Gallagher and D. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Logic Programming: Proc. Eleventh Int. Conf.*, pages 599–613. MIT Press, 1994.

[36] S. Genaim and M. Codish. The Def-inite approach to dependency analysis. In D. Sands, editor, *Proc. Tenth European Symp. Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 2001.

[37] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

[38] J. Howe and A. King. Implementing groundness analysis with definite Boolean functions. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 2000.

[39] N. D. Jones. Concerning the abstract interpretation of Prolog. Draft paper, DIKU, Copenhagen, 1985. Cited in Mellish [52].

[40] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proc. Thirteenth ACM Symp. Principles of Programming Languages*, pages 296–306. ACM Press, 1986.

[41] N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis Horwood, 1987.

[42] A. Kelly, K. Marriott, H. Søndergaard, and P. Stuckey. A practical object-oriented analysis engine for constraint logic programs. *Software—Practice and Experience*, 28(2):199–224, 1998.

[43] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[44] G. Levi and D. Ramundo. A formalization of metaprogramming for real. In D. S. Warren, editor, *Logic Programming: Proc. Tenth Int. Conf.*, pages 354–373. MIT Press, 1993.

[45] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[46] M. J. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In E. Lusk and R. Overbeek, editors, *Logic Programming: Proc. North American Conf. 1989*, pages 963–980. MIT Press, 1989.

[47] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 733–748. MIT Press, 1988.

[48] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. X. Ritter, editor, *Information Processing 89*, pages 601–606. North-Holland, 1989.

[49] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(2 & 3):181–204, 1992.

[50] B. Martens and D. De Shreye. Why untyped nonground metaprogramming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.

[51] C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[52] C. S. Mellish. Abstract interpretation of Prolog programs. In E. Shapiro, editor, *Proc. Third Int. Conf. Logic Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 463–474. Springer-Verlag, 1986.

[53] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2&3):315–347, 1992.

[54] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Scotland, 1981.

[55] R. Ramakrishnan and J. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[56] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 442–453. ACM Press, 1994.

[57] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[58] H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 1986.

[59] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, second edition, 1994.

[60] J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

[61] M. van Emden and R. Kowalski. The semantics of logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.

[62] W. Winsborough. *Automatic, Transparent Parallelization of Logic Programs at Compile Time.* PhD thesis, University of Wisconsin-Madison, Wisconsin, 1988.

[63] J. Wunderwald. Memoing evaluation by source-to-source transformation. In M. Proietti, editor, *Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 1995.