AN APPLICATION OF ABSTRACT INTERPRETATION OF LOGIC PROGRAMS: OCCUR CHECK REDUCTION[1]

Harald Søndergaard
Dansk Datamatik Center
Lundtoftevej 1C, DK-2800 Lyngby, Denmark

Abstract: The occur check in Robinson unification is superfluous in most unifications that take place in practice. The present paper is concerned with the problem of determining circumstances under which the occur check may be safely dispensed with. The method given draws on one outlined by David Plaisted. The framework, however, differs in that we systematically apply the abstract interpretation principle to logic programs. The aim is to give a clear presentation and to facilitate justification of the soundness of the method.

1. Introduction

In the present paper we study the application of *abstract interpretation* to the well-known *occur check problem* of logic programming. Our main interest lies in utilizing the abstract interpretation approach for the purpose of improving Prolog compilers.

One may think of a Prolog program computation as a sequence of unifications mgu(s, b, a), where s is some (current) substitution, while a and b are atomic formulas. Typically, we let *a* denote the left-hand side of a clause, while *b* occurs in some right-hand side, and mgu(s, b, a) yields the most general unifier of s(b) and a. The principal effect of Prolog compilation following the principles of (Warren 1977) is the translation of each positive literal, a, into code doing mgu(s, b, a) for all possible s and b.

Abstract interpretation may provide more information about the diversity of s(b)-variants that can actually occur at a given program point, and an optimizing compiler may capitalize on that to generate more specialized target code. Particular applications of this idea include generation of mode declarations (Mellish 1981) and detection of atom determinacy (Mellish 1985). One may even imagine applications to determine the best storage management policy or backtracking behaviour of a given program.

2. Abstract Interpretation of Logic Programs

The principle of abstract interpretation is rather well established in the case of imperative languages (Cousot 1977). In this section we outline the idea of abstract interpretation as applied to logic programs.

A logic program shall be understood to denote a computation in the universe of substitutions. It consists of a finite number of definite clauses, together with a goal. We introduce the notion of *program points* as indicated by boxes in Figure 1. These alternate with negative literals, such that the right-hand side of a clause has m atoms ($m \geq 0$) and m+1 program points for some m. For any fixed program, we let P denote the set of program points, and $p_{ij} \in P$ denotes the program point immediately after b_{ij}, the j'th atom ($j \geq 0$) in the right-hand side of the i'th clause.

[1] This work was done at Datalogisk Institut, Copenhagen University

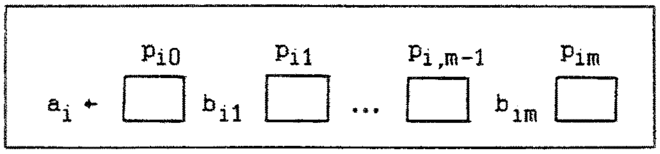


Figure 1: A clause with program points

By substitutions we mean idempotent, almost-identity mappings $s: S = X \rightarrow H(X)$ from the set of variables X to the set of terms $H(X)$. The set of such mappings is pre-ordered by the usual "more general than" relation, $\leq$, defined by $s_1 \leq s_2$ iff $\exists s \in S.\ ss_1 = s_2$. An equivalence relation, $\sim$, on S is defined by $s_1 \sim s_2$ iff $s_1 \leq s_2 \wedge s_2 \leq s_1$. By this, equivalent substitutions are equal modulo consistent renaming of variables. We shall use $\leq$ also in the sense of $\leq/\sim$. The equivalence classes are partially ordered by $\leq$, and when an artificial top element, fail, is added, they form a complete lattice (Eder 1985). This would not be so if we had allowed non-idempotent mappings. Robinson's unification algorithm (Robinson 1965) always yields substitutions in our sense.

We may now give the program a slightly different interpretation, namely as denoting the computation of all the possible substitutions that may occur at a given program point. Thus the new universe consists of mappings from program points to sets of substitutions, and we call these mappings *logs*. Formally, a log is a mapping $L: P \rightarrow 2^S$, and we denote the set of logs by L. The collection of substitutions is a natural analogue to sets of *states* in the case of imperative languages, and we shall call the given interpretation for the *collecting semantics*, following (Nielson 1982). Its usefulness for static analysis of imperative programs has been well established.

The computations thus laid down may not terminate. This is our reason for introducing yet another logic program interpretation which includes a universe of imprecise log descriptions. The idea is that we pay the price of possible imprecision in computations in order to guarantee their termination, and we say that we *approximate* the computation of logs. In spite of the imprecision, such an approximating computation may - as we shall see - still yield useful information. The exact design of the objects used as approximations depends on the purpose of the analysis, that is, what program properties we want to expose. So one can have many different approximating interpretations. In Section 5 we describe *A-logs* - the approximations used for our purpose.

In order to utilize resulting approximations, we must be able to interpret them rigorously. So, a well-defined correspondence between logs and A-logs has to be stated. This is done formally by giving a pair of adjoined mappings, the so-called *abstraction* and *concretization* functions (Cousot 1977). The operators defined by the various interpretations must respect this correspondence, at least so far as to render the analysis sound, i.e. to guarantee that only *safe* approximations are created. In this sense the correspondence induces the abstract interpretation, whence the latter is sometimes called an *induced semantics* (Nielson 1982). We return to these topics in Section 7.

3. The Occur Check Problem

An essential part of Robinson's unification algorithm is the binding of variables x to terms t in order to generate substitution components $x \mapsto t$. Before doing so, however, it must be checked that x is not itself a constituent of t, since in that case unification should fail. The fact that such continual term checking is time consuming constitutes the *occur check problem*. The usual solution is to omit the check, thereby allowing for *circular bindings*. These, however, may cause unwanted behaviour of programs, such as non-termination due to attempts to dereference a circularly bound variable, or acceptance of theorems not true in the first order predicate calculus. For example, one may use Prolog without occur check to "prove" that

(*) $\forall y\, \exists x.\, Q(x, y)$ implies $\exists x\, \forall y.\, Q(x, y)$

Henceforth, we shall call unification without occur check *shallow unification*, as opposed to *Robinson unification*. It has been argued that in practice - owing to Prolog programmers' style - shallow unifications never create circular bindings. This is in conflict with the author's experience and in any case not a satisfactory argument. What we want is a method to distinguish (at compile time) cases where shallow unification can safely be used, in order that the most appropriate code may be generated.

4. Problem Analysis

We shall analyse the problem of determining potential creation of circular bindings by shallow unification in some detail. We must look for computable, sufficient conditions for the absence of circularity, or dually, necessary conditions for circularity. We assume that we are given negative literal ¬b and positive literal a.

An obvious prerequisite for circularity is the repeated occurrence of some variable in a. For example, in (*), a becomes Q(f(u), u), while b is Q(x, f(x)). The example is depicted in Figure 2. The legend is: There is a node for each variable constituent in b or a - upper nodes stem from b, lower ones from a. A dashed line connects a variable with the variable constituents of some term with which it must unify. A zigzag link indicates a multiple occurrence of a variable.

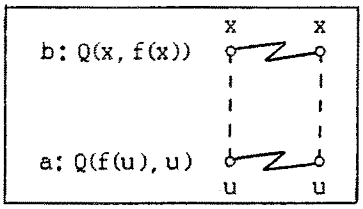


Figure 2: The short circuit indicating possible circularity

Clearly, creation of circularity will reveal itself as a cycle in a graph like this. In a static analysis like ours, however, the cycle may not show itself directly as above. The reason is that nodes correspond to variable constituents of the *original* (sets of) terms in a program while we must take into account cycles created by applying substitutions s to b before unification.

In this more complex case, the cycle above is found to always take one of the two forms shown in Figure 3. The accompanying examples are straightforward, but before we explain the graphs, two definitions are needed. We say that two variables x and y *share* a variable by substitution s if and only if s(x) and s(y) have a variable constituent in common (this is formalized in the next section). And we say that a variable x *spawns* a variable y by substitution s if and only if y is a multiple constituent of s(x).

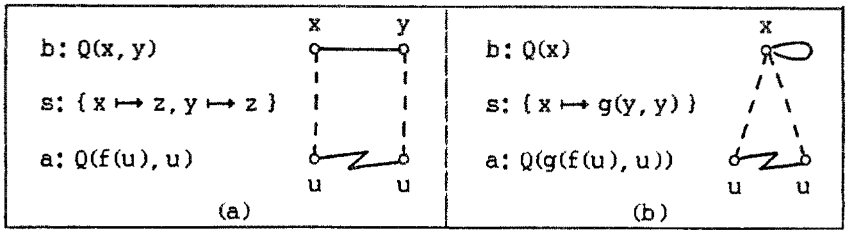


Figure 3: The two paradigms for circularity

In Figure 3(a), x and y share z by s. This is indicated in the graph by a full-drawn edge between (every pair of nodes labeled) x and y. In Figure 3(b), x spawns y by s. This is indicated in the graph by a full-drawn edge from (every node labeled) x to itself.

A cycle now takes the form of a *mixed path* beginning and ending in the same node. A mixed path is defined to be one consisting of the three kinds of edges from above with the following two restrictions: 1) it starts with a dashed edge, and ends in a full-drawn edge (and so is not empty), and 2) it has no two adjacent dashed edges.

The reader may wonder about the reason for distinguishing zigzag ("same") edges on the one hand and full-drawn ("share" or "spawn") edges on the other. The point is that the former are of a temporary nature and will be used only during a test for circularity, whereas the latter ones reflect features of substitutions to be logged and carried around in an approximate form, just as substitutions are propagated in usual computations.

Note that cycles correspond to *potential* generation of circular bindings, rather that guaranteed generation. On the other hand, any circularity will yield a cyclic mixed path in a graph.

5. Approximations

It follows from the above analysis that - in order to serve our purpose - a substitution approximation must convey two kinds of information: First, which pairs of variables *share* by the current substitution, and second, which variables *spawn*. The particular objects that we will use to approximate substitutions are called *A-substitutions*. The set of A-substitutions is denoted by S_A. An A-substitution $s_A \in S_A$ consists of two sets, $G \subseteq \mathbf{X}$ and $E \subseteq \mathbf{X}^2$. Suppose the A-substitution $s_A = (G, E)$ represents a substitution $s\colon \mathbf{X} \to H(\mathbf{X})$. The intended meanings of G and E are:

- for all variables $x \in G$, $s(x)$ is *definitely* a ground term,
- for all pairs $(x,y) \in E$, where $x \neq y$, x and y *may* share a variable by s, while $(x,x) \in E$ indicates that x *may* spawn by s, that is, s(x) may have some multiple variable constituent.

The information kept with G was not found necessary in our analysis, but it proves useful and not too expensive. In the graphs, painted nodes will correspond to G, while full-drawn (non-zigzag) edges correspond to E. Note that E is symmetric but not reflexive.

To every substitution $s \in S$ corresponds a *closest approximation* $\eta(s) \in S_A$. Let *var(t)* denote the *multiset* of variable constituents in term t. Extending (in the natural way) the operators $\cap$ and $-$ to work on multisets, we define the transformer $\eta\colon S \to S_A$ by $\eta(s) = (G, E)$ where

$$G = \{ x \in \mathbf{X} \mid var(s(x)) = \emptyset \}$$

$$E = \{ (x,y) \in \mathbf{X}^2 \mid x \neq y \wedge var(s(x)) \cap var(s(y)) \neq \emptyset \} \cup \{ (x,x) \in \mathbf{X}^2 \mid var(s(x)) - \mathbf{X} \neq \emptyset \}$$

Let $\mathbf{G} = 2^{\mathbf{X}}$ and $\mathbf{E} = 2^{\mathbf{X}^2}$. Then **G** and **E** form complete lattices under usual set inclusion. $\mathbf{S_A} = \mathbf{G} \times \mathbf{E}$ is a complete lattice under the lexical ordering $\leq_A$, defined by

$$(G_1, E_1) \leq_A (G_2, E_2) \text{ iff } G_1 \subset G_2 \vee (G_1 = G_2 \wedge E_1 \subseteq E_2).$$

This ordering is induced by $\leq$ and the demand for a *monotonic* η. We say that the A-substitution s_A *safely approximates* the substitution s if and only if $\eta(s) \leq_A s_A$.

The set of logs, L, as defined in Section 2, forms a lattice under pointwise inclusion and is the universe for the collecting semantics. Our approximate semantics makes use of what we call an *A-log* to associate with every $p \in \mathbf{P}$ a set of A-substitutions. Formally, an A-log is a mapping $L_A: \mathbf{P} \rightarrow 2^{\mathbf{S_A}}$, and we denote the set of A-logs by $\mathbf{L_A}$. Under pointwise inclusion, $\mathbf{L_A}$ forms a lattice, and this is the universe of our abstract interpretation. When restricted to a finite set of variables, the lattice is of finite height. This will be the case in our abstract interpretation, since no variable renaming ever takes place. We later define what it means that a log is safely approximated by an A-log.

6. The Method

We merely give a loose description of the method. It was inspired by those outlined in (Plaisted 1984) and works in two steps as explained below.

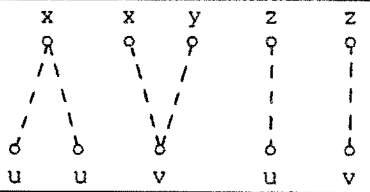
6.1. Preunification

Much of the *approximating unification* can be done once for all. So we apply *preunification* on every pair of atoms (b_{ij}, a_k). The result is either *fail* or an intermediate object T_{ijk} which we call a *template*. Preunification of (b, a) is done in three steps:

1) Robinson unification is applied to the pair.
2) If 1) did not yield fail, the following deterministic rewriting is applied to {(b, a)}: Pairs $(f(t_1,\ldots,t_n), f(t_1',\ldots,t_n'))$ are split into n pairs $(t_1, t_1'),\ldots,(t_n, t_n')$. Since 1) succeeded, an irreducible pair consists of at least one variable.
3) Templates are created by replacing every pair (t, t') by the pair of multisets (var(t), var(t')). Thus templates may be seen as sets of bindings: A variable is bound to the multiset var(t) if it should unify with the term t. An example of a template is given in Figure 4.

b_{ij}: Q(x, f(g(x, y)), g(z, z))

T_{ijk}:



a_k: Q(g(u, u), f(v), g(u, v))

Figure 4: A pair of atoms to be unified and the corresponding template

Note that templates are graphs having only dashed edges. The abstract interpretation consists in applying A-substitutions to templates. The resulting graphs in turn will be manipulated in order to reveal potential circularity and to yield new A-substitutions, as described in the sequel.

For reasons soon to be clear, we actually apply *two* A-substitutions to a template, an *upper* and a *lower* one. Let T be a template, and let s_A and s_A' be A-substitutions. Then $T[s_A, s_A']$ denotes the graph that results from applying s_A to the upper nodes of T, and s_A' to the lower nodes. As already mentioned, we picture these applications by filling out the ground nodes G and adding full-drawn edges for the share and spawn edges E.

6.2. Iteration

After preunification, an iteration process propagates A-substitutions among program points, mimicking a usual computation. There is a crucial difference, though. The essential feature of the approximating computation is what we will call *locality*. It generates A-logs having the property that every A-substitution attached to the program point p_{ij} exclusively has variables *local* to the i'th clause. This is the virtue (not shared by the collecting semantics) that renders the usual renaming of variables superfluous.

There is a price to be paid for this property. In a usual computation, the unification of b_{ij} and a_k provides for all necessary exchange of information between b_{ij} and the k'th clause. The approximating computation, however, forgets about b_{ij} temporarily, owing to the locality. So, before returning an A-substitution from the k'th clause, the relation to b_{ij} must be reestablished, as indicated in Figure 5. That is, the iteration is defined in terms of two kinds of approximating unification, a *forward* one, A-unify↓, and a *backward* one, A-unify↑.

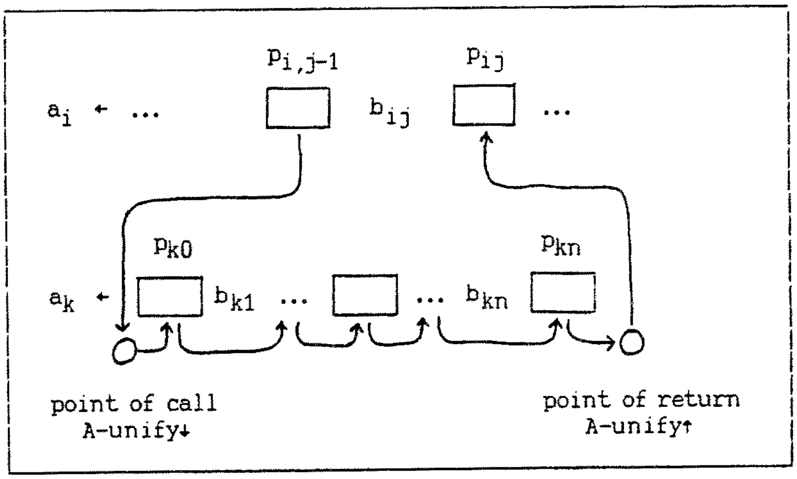


Figure 5: The propagation of A-substitutions

Let $e_A = \eta(e)$ be the empty A-substitution. More precisely then, we compute the least fixed-point of the operator IterA: $\mathbf{L_A} \rightarrow \mathbf{L_A}$, defined by

$$\text{IterA} = \lambda L_A . L_A' \text{ where } (s_A', L_A') = \text{It}(p_{00}, (e_A, L_A)).$$

The operator It: $\mathbf{P} \times (\mathbf{S_A} \times \mathbf{L_A}) \rightarrow \mathbf{S_A} \times \mathbf{L_A}$ provides for the propagation of A-substitutions from a program point p_{ij} to the end of the i'th clause. It carries the current A-substitution and A-log, and is defined recursively by

```
It($p_{i,j-1}$, ($s_A$, $L_A$)) =
  if $p_{ij}$ exists then ($s_A'$, $L_A'$)
    where k is chosen such that the template T = $T_{ijk}$ exists, and

        $s_A'''$              =  A-unify↓($s_A$, T)
        $L_A'''$              =  $L_A[p_{k0} \rightarrow L_A(p_{k0}) \cup \{s_A'''\}]$
        ($s_A''$, $L_A''$)    =  It($p_{k0}$, ($s_A'''$, $L_A'''$))
        $s_A'$                =  A-unify↑($s_A$, $s_A''$, T)
        $L_A'$                =  $L_A''[p_{ij} \rightarrow L_A''(p_{ij}) \cup \{s_A'\}]$
  else ($s_A$, $L_A$)
```

The expression F[x→y] denotes the mapping which acts like F, except that it returns y for x.

6.3. A-unification

The two kinds of A-unification are loosely described in terms of five primitive operators. We leave out an exact description of these, relying on the explanatory power of the eight examples given in Figure 6. Suffice it to say that

- "Upper" extracts the upper A-substitution from a graph,
- "Lower" extracts the lower A-substitution from a graph,
- "TransG" transfers groundness information in a graph,
- "TransE" transfers share and spawn information in a graph, while
- "TestC" tests a graph for circularity. It works by adding temporary zigzag edges between nodes having the same labels (and being both upper or both lower). It then detects potential circularity by finding possible mixed paths forming cycles in the graph.

We may now define the A-unifications:

$$\text{A-unify}\downarrow (s_A, T) = \text{Lower (TransE (TestC (TransG (T[} s_A, e_A \text{]))))}$$

$$\text{A-unify}\uparrow (s_A, s_A', T) = \text{Upper (TransE (TransG (T[} s_A, s_A' \text{])))}$$

The order of applying the primitive operations is crucial. So is the fact that circularity tests are performed only in forward A-unifications.

We now turn to the examples. In all cases we assume that we are performing forward A-unification. The initial graph appears just after application of A-substitutions. The intermediate graph is the (temporary) result of TestC. The final graph is ready for A-substitution extraction by Lower.

1) Preunification fails, so no template was ever created.

2) The template becomes very simple; no circular bindings can be created.

3) We are given that y is ground. TransG will transfer ground information to the extent of painting all nodes. TestC will realize that no circular bindings may be created (or rather, they may, temporarily, but in that case even shallow unification will ultimately fail). Lower will extract a lower A-substitution having G-component {u, v} to indicate that u and v from here on represent ground terms.

4) We are given that x and y may share. TestC finds no circularity, TransE will find that sharing may apply to u and v as well, and Lower will extract that information by yielding an A-substitution with E-component {(u, v)} (E is understood to be symmetric). Note that *final* graphs can never indicate circularity, and in this example, circularity will not be signaled on return either, since the circularity test is not part of a backward A-unification.

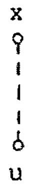
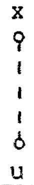
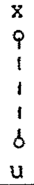
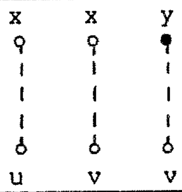
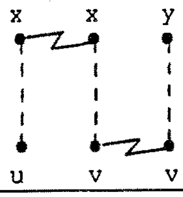
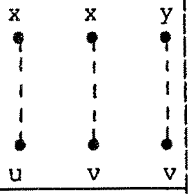
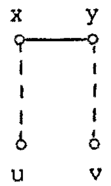
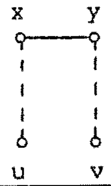
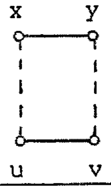
	Atoms	Initial graph	Interm. graph	Final graph
1	b: Q(g(f(x), g(y, z))) s_A: e_A a: Q(g(u, f(u)))	(fail)		
2	b: Q(x, x) s_A: e_A a: Q(u, u)			
3	b: Q(x, x, y) s_A: ({y}, ∅) a: Q(u, v, v)			
4	b: Q(x, f(y)) s_A: (∅, {(x,y)}) a: Q(u, v)			

Figure 6: Sample graphs

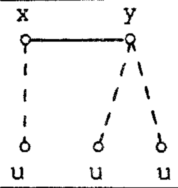
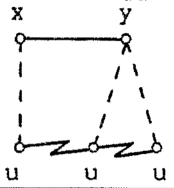
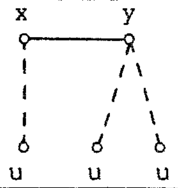
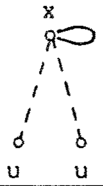
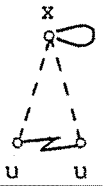
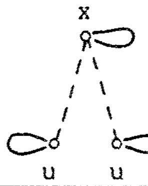
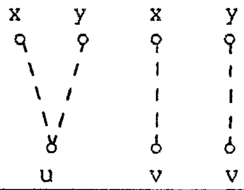
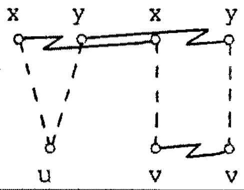
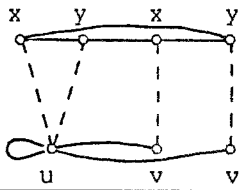
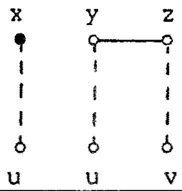
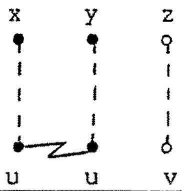
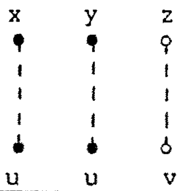
	Atoms	Initial graph	Interm. graph	Final graph
5	b: Q(x, y) s_A: (∅, {(x,y)}) a: Q(u, g(u, u))			
6	b: Q(x) s_A: (∅, {(x,x)}) a: Q(g(u, u))			
7	b: Q(g(x, y), x, y) s_A: e_A a: Q(u, v, v)			
8	b: Q(x, g(y, z)) s_A: ({x}, {(y,z)}) a: Q(u, g(u, v))			

Figure 6: Sample graphs, continued

5) We are given an A-substitution indicating that x and y may share a variable. TestC finds circularity possible in this case.

6) We are given that x may spawn a variable. TestC reveals that circularity is possible. Also, TransE will record that u may spawn a variable. Lower will extract an A-substitution containing precisely that piece of information, i.e. having as E-component {(u, u)}.

7) TestC finds no circularity since the cycle in the intermediate graph is not a mixed path. However, TransE will find that u and v may share a variable and that, furthermore, the former may spawn a variable. Lower will indicate this by extracting as E-component {(u, u), (u, v)}.

8) x is initially ground, and y and z are suspected to share a variable. TransG will mark u and y nodes as ground and subsequently delete the (y,z) edge. TestC finds no circularity. This is a forward A-unification, so Lower yields an A-substitution having G-component {u}. On return, it will effectively be concluded that y and z cannot share.

Note that the method also yields useful groundness information which may be used for optimizing purposes. In a way, circularity information just comes out as a sort of by-product.

As an example of the limitations of the method we give the simplest program for which it will *not* be detected that occur checks are unnecessary:

```
        ←  Q(x, x)
Q(y, z) ←  R(y, z)
R(u, u) ←
```

7. Soundness

We now lay down what it means for the method to be sound with respect to the collecting semantics. Note that the collecting semantics can be given in the style of IterA of Section 6.2. We just change the universe from $\mathbf{L_A}$ to $\mathbf{L}$ and give the primitive operators a different interpretation: A-unify↓ then denotes Robinson unification, while A-unify↑ denotes the identity function on S. In accordance with this interpretation, the templates will constitute the initial connection graph (Kowalski 1975) of the program.

In Figure 7 we show the involved universes. In all cases these are complete lattices. The orderings on logs and A-logs are pointwise inclusions:

$$L \sqsubseteq L' \text{ iff } \forall p \in \mathbf{P}.\ L(p) \subseteq L'(p)$$
$$L_A \sqsubseteq_A L_A' \text{ iff } \forall p \in \mathbf{P}.\ L_A(p) \subseteq L_A'(p)$$

"Collecting" world				"Approximating" world
Substitutions	$(\mathbf{S}, \leq)$		$(\mathbf{S_A}, \leq_A)$	A-substitutions
Sets of substitutions	$(2^{\mathbf{S}}, \subseteq)$	$\underset{\gamma}{\overset{\alpha}{\rightleftarrows}}$	$(2^{\mathbf{S_A}}, \subseteq)$	Sets of A-substitutions
Logs	$(\mathbf{L}, \sqsubseteq)$		$(\mathbf{L_A}, \sqsubseteq_A)$	A-logs

Figure 7: The central universes

Whereas the lattices on the left have height ω, the heights of the approximating ones are all finite, provided the set of variables $\mathbf{X}$ is. We take the *abstraction function*, α, and the *concretization function*, γ, to have types as given by Figure 7 and define

$$\alpha(S) = \{\, \eta(s) \mid s \in S \,\}$$
$$\gamma(S_A) = \{\, s \in \mathbf{S} \mid \eta(s) \in S_A \,\}$$

These are monotonic and fulfil the requirements (Cousot 1977) that

$$S \subseteq \gamma(\alpha(S)) \quad\text{and}\quad S_A = \alpha(\gamma(S_A))$$

for all $S \subseteq \mathbf{S}$ and $S_A \subseteq \mathbf{S_A}$. An A-log L_A safely approximates (s.a.) a log L if and only if it does so pointwise:

$$L_A \text{ s.a. } L \text{ iff } \forall p \in \mathbf{P}.\ L_A(p) \text{ s.a. } L(p)$$

A set of A-substitutions S_A safely approximates a set of substitutions S if and only if every substitution in S is safely approximated by some A-substitution in S_A:

$$S_A \text{ s.a. } S \text{ iff } \forall s \in S\ \exists s_A \in S_A .\ s_A \text{ s.a. } s$$

Thus we accept a slip of exactness in two dimensions: First, the set of A-substitutions may be too large, and second, the approximation of a particular substitution may not be closest. In the case of substitutions we have as already mentioned:

$$s_A \text{ s.a. } s \text{ iff } \eta(s) \leq_A s_A$$

In general then, to verify soundness, we must show for corresponding operators $\Omega: D \longrightarrow D$ and $\Omega_A: D_A \longrightarrow D_A$, working on corresponding universes D and D_A, that Ω_A safely approximates Ω. We define

$$\Omega_A \text{ s.a. } \Omega \text{ iff } \forall d \in D\ \forall d_A \in D_A .\ \ (d_A \text{ s.a. } d \Rightarrow \Omega_A(d_A) \text{ s.a. } \Omega(d)) .$$

In other words, we demand that the operators respect the s.a. relation.

8. Conclusion

We have described a rather elaborate compile time analysis. The reader may be left with the impression that we propose to do at compile time all the work we wanted to avoid at run time, or maybe more. This is not so. The point to be stressed is this: Though possibly elaborate, even very simple compile time transformations may shorten the run time strikingly, to the effect of bringing the total time far below that used for an interpretation. This was firmly demonstrated in practice by (Jones 1985) who investigated the applicability of *partial evaluation* for compiling and compiler generation purposes. From this point of view, Prolog compilation essentially consists in partial evaluations of the unification procedure with respect to the positive literals.

No analysis of the algorithm's time complexity has been undertaken. Clearly, the critical quantity is the number of variables in a clause, so the worst case complexity should be measured in terms of V, the largest number of variables in a clause of the program. The complexity is conjectured to be exponential in V. This does not say much, though, since V is usually quite small and independent of the program's size.

The efficiency is thus considered an empirical question. If it is too low, a more crude method is easily obtained by taking least upper bounds rather than *collecting* sets of A-substitutions. In this case, we conjecture that the time complexity is polynomial in V.

A whole family of still more intricate methods for detecting circularity was sketched in (Plaisted 1984). The aim was to give a *complete* solution in the following sense: If no circularity could actually occur, then there should always be one method laborious enough to determine this. The present method draws heavily on Plaisted's but differs from those in that it

- is simpler, owing to the lack of generality,
- is recognizable as an abstract interpretation. In particular, an approximating universe is laid down,
- behaves better even in trivial cases like Example 2 above, owing to the use of preunification,
- is guaranteed to terminate.

The method has not yet been implemented. It seems feasible to express other kinds of program analysis as abstract interpretations. In fact, the iteration procedure of Section 6.2 would seem quite a practicable mould for giving a solution to the mode declaration problem.

9. References

Cousot, P. and R. Cousot,
Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc. 4th ACM POPL Symp.*, Los Angeles, California (June 1977) 238-252

Eder, E.,
Properties of substitutions and unifications, *Journal of Symbolic Computation* 1,1 (March 1985) 31-46

Jones, N. D., P. Sestoft and H. Søndergaard,
An experiment in partial evaluation: The generation of a compiler generator, in *LNCS* 202: *Rewriting Techniques and Applications* (ed. J.-P. Jouannaud), Springer Verlag (1985) 124-140

Kowalski, R.,
A proof procedure using connection graphs, *Journal of the ACM* 22,4 (October 1975) 572-595

Mellish, C. S.,
The Automatic Generation of Mode Declarations for Prolog Programs, DAI Research Paper no. 163, University of Edinburgh, Scotland (1981)

Mellish, C. S.,
Abstract Interpretation of Prolog Programs, extended abstract presented at the Workshop on Abstract Interpretation of Declarative Languages, Canterbury, England (August 1985)

Nielson, F.,
A denotational framework for data flow analysis, *Acta Informatica* 18 (1982) 265-287

Plaisted, D.,
The occur-check problem in Prolog, *Proc. Intl. Symp. Logic Programming*, Atlantic City, New Jersey (February 1984) 272-280

Robinson, J. A.,
A machine-oriented logic based on the resolution principle, *Journal of the ACM* 12,1 (January 1965) 23-41

Warren, D. H. D.,
Implementing Prolog - Compiling Predicate Logic Programs, DAI Research Report no. 39, University of Edinburgh, Scotland (1977)