PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

AN ABSTRACT INTERPRETATION SCHEME FOR LOGIC PROGRAMS BASED ON TYPE EXPRESSION

Arvind K. Bansal [1] and Leon Sterling

Department Of Computer Engineering & Science
Case Western Reserve University
Cleveland, OH 44106, USA

Abstract

This paper describes a scheme for the abstract interpretation of logic programs based on type information. Our scheme has four components: generalization, abstract unification, summarization and concretization. Algorithms for generalization, abstract unification, and summarization are discussed. Our scheme does not suffer from the aliasing problem. Our main application of abstract interpretation is deriving producer-consumer relationships and classification of procedures for transformation of logic programs for efficient execution under committed-choice AND-parallelism extended to find all solutions. However, our scheme is general enough to be directly applied for type generation, compile time memory allocation and efficient unification.

Keywords: Abstract interpretation, abstract unification, logic programming, mode analysis, polymorphic types, type generation

1 Introduction

Abstract interpretation is the simulation of program behavior in some abstract domain. In recent years, abstract interpretation has emerged as a powerful tool to derive properties of logic programs. Mellish (1986) used abstract interpretation for efficient compilation and unification in logic programs. Bruynooghe et. al. (1987) incorporated polymorphic type information in the abstract domain for compile time memory allocation, garbage collection, and efficient unification.

Our motivation for working on abstract interpretation is to detect the inherent parallelism in logic programs at compile time. Mode information is automatically derived which guides the transformation of programs for efficient execution under an extension of the committed-choice AND-parallel model which finds all solutions to goals (see Bansal and Sterling 1987 & 1988).

The abstract domain we selected for deriving this information is type expressions. Logic programming languages exhibit both parameteric polymorphism (Zobel 1987, Mishra and Reddy 1985, Mycroft and O'Keefe 1984, Bruynooghe et. al. 1987, Bansal 1988) and inclusion polymorphism (Bansal 1988). Mode information has been derived in terms of type expressions. Integration of mode information with type information makes modes more expressive.

In this paper, we introduce a scheme for abstract interpretation. Our abstract interpretation scheme has four components: generalization, abstract unification, summarization and concretization. We propose a general algorithm for abstract unification which handles inclusion polymorphism and equivalence classes formed by the unification of two variables. The aliasing problem (Debray 1986), present in Mellish's scheme (Mellish 1986), is absent in ours. Our scheme takes care of negation, built-in predicates, and recursive data structures formed top down or bottom up, by self-recursive or mutually recursive predicates.

Using abstract interpretation, we have derived mode information for various predicates needed for producer-consumer relationship, automatic detection of a major class of guards for efficient pruning of the search space, derivation of type information for effective memory allocation and unification, and automatic classification of procedures for compilation of logic programs (including multiple solutions) to committed choice AND-parallelism (Bansal 1988). This scheme has been successfully implemented using Quintus Prolog on a Sun 3/60.

The organization of the paper is as follows. The next chapter discusses our abstract domain based on polymorphic types. Section 3 discusses concepts and algorithms for generalization, abstract unification, summarization, concretization. Section 4 gives an overall abstract interpreter, discussing recursive data structures, negation, and built-in predicates. The last two sections compare our work with related work and give conclusions respectively.

[1] Author's current address:
Department of Mathematical Sciences,
Kent State University, OH 44242, USA

2 Abstract Domain

Our abstract domain restricts the standard domain by identifying sets of terms as types and reasoning with the types. We assume familiarity with polymorphic types specifically parametric and inclusion polymorphism. A good reference is available in (Cardelli and Wegner 1985)

The elements of our abstract domain are type variables, basic types, and type expressions. A grammar for type expressions is given below.

Exp ::= *Basic* | < *functor* >(Exp_1, ...,Exp_m) |
$Exp_1 \cup Exp_2$ | *List* | *Tuple* | *nil*
Basic ::= $\mathcal{Z}$ | $\mathcal{C}$ | μ | ϕ | *Type-var*
List ::= [*Exp* | *List*] $\cup$ *nil*
Tuple ::= (*Exp*, *Tuple*) $\cup$ *Exp*

A **type variable** is a *generic symbol*, which can be associated with any *type expression*. It will be denoted by small Greek letters α, β, γ, δ.

Our **basic types** are *integer*, *atomic-symbol*, *universal type*, and *null type*. We denote *integer* by the calligraphic letter $\mathcal{Z}$, *atomic-symbol* by the calligraphic letter $\mathcal{C}$, *universal type* by μ, and *null type* by ϕ. We use *nil* as the bottom value for all structures. Its use is a special case of parametric polymorphism known as value sharing (Cardelli and Wegner 1985).

A **type expression** is defined recursively. It is a type variable, a basic type, a compound term consisting of a functor of arity n with n arguments as type expressions, or a union of type expressions, used to incorporate additive polymorphism (see Mishra and Reddy 1985). Type expressions are denoted by ω, σ, τ. A union of type expressions is denoted by $\{\tau_1,..., \tau_m\}$, where m $\geq$ 2. For example $\mathcal{Z}$, $\mathcal{C}$, $\{\alpha, \mathcal{C}\}$, $f(\mathcal{Z}, \alpha)$ are type expressions. A list of any type expression τ is denoted by τ^* throughout this paper. For example, a list of integers is denoted $\mathcal{Z}^*$.

Recursive type expressions, such as $\mathcal{Z}^*$, need to be handled specially by abstract interpretation. A different notation is used and explained in Section 4. One concept that will be needed is the periodicity of a recursive type expression. A **common subexpression** of a recursive type expression is the biggest repeating homogeneous subexpression. For example, the type expression for a list of an even number of integers, denoted *even* $\mathcal{Z}^*$, has common subexpression $\mathcal{Z}$. The expression *odd* $\mathcal{Z}^*$ denotes a list of an odd number of integers. The **perodicity of a recursive type expression** is the number of common subexpressions present in one cycle of the recursive type expression. For example, $\mathcal{Z}^*$ has periodicity 1 and *even* $\mathcal{Z}^*$ has periodicity 2.

Two type expressions having the same common subexpressions and periodicity of the form m and k*m, where m, k are positive integers, exhibit inclusion polymorphism. For example, $\mathcal{Z}^*$ includes the type expression *even* $\mathcal{Z}^*$.

Type expressions are manipulated during abstract interpretation. Our terminology for handling type expressions follows. An **abstract substitution element** is a binding of a type variable α with a type expression τ. It will be denoted as α/τ. An **abstract substitution**, denoted Θ, is a finite set of abstract substitution elements.

An **abstract instance** of an abstract term τ is obtained by applying an abstract substitution Θ on it. It will be denoted as $\tau \circ \Theta$. For example, if $\tau = b(\alpha, \beta)$ and $\Theta = \{\alpha/\mathcal{Z}, \beta/\mathcal{C}\}$ then the abstract instance is given by $\tau \circ \Theta = b(\mathcal{Z}, \mathcal{C})$.

We need mode information for classification of procedures for transformation to committed-choice AND-parallel programs (Bansal 1988). Abstract interpretation is used for deriving mode information (Bansal 1988, Bruynooghe et. al. 1987, Bruynooghe and Jenssens 1988, Mellish 1986). We distinguish four modes, given below:

The **calling mode of a goal** G is the abstract instance, represented in terms of type expressions, of the abstract terms in the abstract domain of the goal before the goal execution. It will be denoted by G^C.

The **success mode of a goal** G is the abstract instance, represented in terms of type expressions, of the abstract terms in the abstract domain of the goal after the goal execution. It will be denoted by G^S.

The **initial mode of a clause** A is the abstract instance, represented in terms of type expressions, of the abstract terms in the clause before the abstract interpretation of A. It will be denoted by A^I.

The **final mode of a clause** A is the abstract instance, represented in terms of type expressions, of the abstract terms in A after the abstract interpretation of A. It will be denoted by A^F.

3 Basic Elements

The essence of abstract interpretation is the construction and traversal of an AND-OR tree in the abstract domain. The abstract interpreter has four basic components. *Generalization* maps the query and program into an abstract goal and abstract set of clauses respectively. *Abstract unification* is the abstract domain analogue of unification in the standard domain and computes the most general abstract instance of two terms in the abstract domain. *Summarization* combines the results of alternative 'abstract computations'. Finally, *concretization* associates the variables of the predicates with the abstract bindings achieved using abstract interpretation. This section describes each stage in detail.

3.1 Generalization

Generalization is a mapping of logical terms in the standard domain to an abstract term, given as a type ex-

pression, in the abstract domain. It is essentially the operation *abstraction* given by Bruynooghe and Jenssens (1988).

The mapping replaces integers by $\mathcal{Z}$ and atomic symbols by $\mathcal{C}$. Variables in program clauses are replaced by type variables, while variables in the query need user qualification. Variables representing universal types are replaced by μ. The functor names of the standard terms, list constructs and tuple constructs are unaltered. For example, the term $a([1, 2], john)$ maps to $a(\mathcal{Z}^*, \mathcal{C})$.

3.2 Abstract Unification

Abstract unification is an abstract domain analogue of unification in standard domain. We refer to the latter as *standard unification*. Abstract unification takes into account inclusion polymorphism. We introduce some new concepts required for abstract unification and then present the algorithm.

Two abstract terms τ and ω are **abstract equivalent** if

(1) τ and ω are syntactically identical; **(2)** τ is included in ω through inclusion polymorphism; or **(3)** τ and ω are functors having the same functor name and their arguments are abstract equivalent. For example, $f(\mathcal{C}, g(\mathcal{Z}))$ and $f(\mathcal{Z}, g(\mathcal{C}))$ are abstract equivalent.

Note that abstract equivalence is not actually an equivalence relation, due to inclusion polymorphism. If two distinct types τ_1 and τ_2 are included in some type ω, then τ_1 and τ_2 are both abstract equivalent to ω, but are not themselves abstract equivalent. Thus abstract equivalence is not transitive.

Two abstract terms Q and R abstract unify if there exists *abstract instances* Q_1 of Q and R_1 of R such that Q_1 and R_1 are abstract equivalent. The notion of most general common instance of two terms in the standard domain can be adapted to define the most general common abstract term, (which we will denote **mgcat**) included in two abstract terms. The complication is handling inclusion polymorphism. If two terms abstract unify then there exists an abstract substitution Θ such that $Q\circ\Theta$ and $R\circ\Theta$ are *abstract equivalent*.

For example, the abstract terms $a(\alpha, \mathcal{Z}, \mathcal{C})$ and $a(\mathcal{C}, \mathcal{C}, \mathcal{Z})$ abstract unify with *mgcat* as $(\mathcal{C}, \mathcal{Z}, \mathcal{Z})$. The corresponding abstract substitution is $< \alpha/\mathcal{C} >$ and abstract instances $a(\mathcal{C}, \mathcal{Z}, \mathcal{C})$ and $a(\mathcal{C}, \mathcal{C}, \mathcal{Z})$ are abstract equivalent.

Two abstract terms τ and ω abstract unify if
(1) τ and ω are abstract equivalent
(2) τ and ω are ordered sets of type expressions of the form $\{\tau_1, ..., \tau_m\}$ and $\{\omega_1, ..., \omega_n\}$ and there exists a nonempty maximal ordered set of type expressions $M = \{\tau_1, ..., \tau_k, \omega_1, ..., \omega_l\}$ ($k \leq m$ and $l \leq n$) such that every $\tau_i \in M$ is abstract equivalent with at least one element $\omega_j \in M$; similarly, every $\omega_j \in M$ is abstract equivalent with at least one element $\tau_i \in M$
(3) τ or ω (or both) are recursive type expressions, and a finite unfolding of the recursive type expressions makes the unfolded parts *abstract equivalent*. For example, $\mathcal{Z}^*$ and *even* $\mathcal{Z}^*$ abstract unify with *mgcat* as *even* $\mathcal{Z}^*$.

There are three differences of abstract unification from standard unification. Firstly, the abstract terms may be a set of expressions. In standard unification, a variable always has a single value. Secondly, one abstract term may be included in the other through inclusion polymorphism. In standard unification, two ground logical terms must be identical. Finally, abstract unification handles recursive data structures with different periodicity.

The **abstract disagreement set**, D, of two abstract terms τ and ω is the pair of abstract subterms D_τ and D_ω occuring at the same symbol position, such that there is at least one symbol in D_τ which is not identical with the corresponding symbol in D_ω. This definition is different from the definition of disagreement set given by Lloyd (1984) because the position of the symbol is picked up nondeterministically without any ordering. For example, if $\tau = a(\alpha, \alpha)$ and $\omega = a(\mathcal{Z}, \mathcal{C})$ then there are two disagreement sets namely $D_1 = \{\alpha, \mathcal{Z}\}$ and $D_2 = \{\alpha, \mathcal{C}\}$.

Singular composition computes the greatest lower bound of the abstract substitutions of the same variable α. It is *commutative* and applied to compute the abstract substitution element for α present in different abstract disagreement sets during abstract unification. It is denoted by the symbol $\oplus$.

Consider two disagreement sets $D_1 = \{\alpha, \mathcal{Z}\}$ and $D_2 = \{\alpha, \mathcal{C}\}$. The singular composition $D_1 \oplus D_2$ gives the abstract binding $\alpha/\mathcal{Z}$, since $\mathcal{Z}$ is included in $\mathcal{C}$. A procedure for singular composition is given in Figure 1.

The procedure given in Figure 1 may not terminate when both abstract substitution elements are recursive type expressions which are not abstract unifiable. For example, *even* $\mathcal{Z}^*$ and *odd* $\mathcal{Z}^*$ are not abstract unifiable resulting in the non-termination of the singular composition procedure. A memo function is used to turn the procedure into an algorithm, by detecting infinite loops while finding out the abstract unifiability of recursive type expressions.

Nonsingular composition is similar to composition as defined by Lloyd (1984) for standard unification. The difference between is due to the presence of sets of bindings, recursive data structures and inclusion polymorphism in abstract domain. The nonsingular composition of the abstract substitutions
$\Theta_1/< \alpha_1/\tau_1,..., \alpha_m/\tau_m >$ and $\Theta_2/< \beta_1/\omega_1, ..., \beta_m/\omega_m >$
is obtained from the ordered set
$< \alpha_1/\tau_1 \circ \Theta_2,..., \alpha_m/\tau_m \circ \Theta_2, \beta_1/\omega_1, ..., \beta_m/\omega_m >$
by deleting bindings in $\tau_i \circ \Theta_2$ which are of the form

α_i with the exception of recursive type expressions and removing bindings β_i/ω_i such that $\beta_i \in \{\alpha_1, ..., \alpha_m\}$. Note that $\tau_i \circ \Theta_2$ may be a set of expressions. Nonsingular composition is denoted by $\odot$.

For example, if $\Theta_1 = < \alpha/\alpha_1, \beta/\mathcal{Z} >$ and $\Theta_2 = < \alpha_1/\{h(\mathcal{Z}), \mathcal{C}\}, \beta/\mathcal{C} >$. The nonsingular composition, $\Theta_1 \odot \Theta_2$, is $< \alpha/\{h(\mathcal{Z}), \mathcal{C}\}, \beta/\mathcal{Z} >$.

Procedure Singular Composition;
Input: Two abstract substitution elements of the form α/τ and α/ω;
Output: Abstract substitution element α/σ;
begin
if τ (or ω) $= \mu$ **then** $\sigma := \omega$ (or τ)
elseif ω or τ is ϕ **then** $\sigma := \phi$
elseif τ (or ω) is a basic type and τ (or ω) includes ω (or τ) **then** $\sigma := \omega$ (or τ)
elseif τ, ω are recursive type expressions with common subexpression δ, and periodicities m, n **then**
$\sigma :=$ a recursive type expression with common subexpression δ with periodicity equal to lcm(m, n)
elseif τ and ω are abstract unifiable, and one is nonrecursive while the other is a recursive type expression, **then**
$\sigma :=$ *mgcat* of τ and ω
elseif τ is an ordered set $\{\tau_1, ..., \tau_m\}$ and ω is an ordered set [2] $\{\omega_1, ..., \omega_n\}$ **then**
begin
initialize S to $\emptyset$.
for each pair $(\tau_i, \omega_j) \in \tau \times \omega$ **do**
$S := S \bigcup \{\tau_i \oplus \omega_j\}$;
Let S be of the form $\{t_1, ..., t_n\}$;
for each element $t_i \in S$ **do**
if t_i is included in $t_j \in$ S (i $\neq$ j) through inclusion polymorphism **then** $S := S$ - $\{t_i\}$;
$\sigma :=$ S
end
else $\sigma := \phi$
end.

Figure 1: Procedure for singular composition

To abstract unify two abstract terms τ and ω, compute the abstract disagreement set $D = (D_\tau, D_\omega)$. If D has no unbound type variables and both D_τ and D_ω are not abstract equivalent then abstract unification fails, otherwise D_τ and D_ω are replaced by the *mgcat* of D_τ and D_ω. If either D_τ or D_ω has an unbound type variable α then all the disagreement sets are found for α. The singular composition of these disagreement sets is found to determine the abstract binding δ for α. Singular composition of all the disagreement sets is necessary to determine the greatest lower bound. After the abstract binding is found, the previous abstract substitution Σ is updated by forming the nonsingular composition $\Sigma\odot < \alpha/\delta >$. This new abstract substitution is applied on the two abstract terms to form new abstract terms $\tau \circ \Sigma\odot < \alpha/\delta >$ and $\omega \circ \Sigma\odot < \alpha/\delta >$.

[2] If necessary, consider a single element as a set

This process is continued until the disagreement set is empty.

Consider the abstract unification of two abstract terms $a(\alpha, \alpha)$ and $a(\mathcal{Z}, \mathcal{C})$. There are two disagreement set $D_1 = \{\alpha, \mathcal{C}\}$ and $D_2 = \{\alpha, \mathcal{Z}\}$. The abstract substitution $< \alpha/\mathcal{C} >$ obtained from D_1, when applied to first term gives the abstract instance as $a(\mathcal{C}, \mathcal{C})$ which abstract unifies with the second term $a(\mathcal{Z}, \mathcal{C})$ to give the *mgcat* $a(\mathcal{Z}, \mathcal{C})$ which is wrong. The correct *mgcat* is $a(\mathcal{Z}, \mathcal{Z})$ which is achieved by taking the singular composition of the binding $< \alpha/\mathcal{C} >$, obtained from D_1 and $< \alpha/\mathcal{Z} >$ obtained from D_2.

Singular composition is not present in standard unification because of the absence of inclusion polymorphism, recursive type expressions, and sets of type expressions. In standard unification, there is only one compatible binding.

The algorithm for abstract unification is given in Figure 2.

Algorithm Abstract Unification;
Input: Two abstract terms τ and ω;
Output: mgcat δ and the corresponding abstract substitution Σ;
begin
Initialize Σ to ϵ (the identity abstract substitution);
while τ is not identical to ω **do**
begin
Compute an abstract disagreement set $D = \{D_\tau, D_\omega\}$;
if D does not have any unbound type variable **then**
if D_τ and D_ω are abstract equivalent **then**
begin
Find the *mgcat* I of D_τ and D_ω;
Replace D_τ and D_ω by I in τ and ω respectively
end
else fail and return $\Sigma := \emptyset$ and $\delta := \phi$
elseif D contains an unbound type variable α **then**
begin
Find all the abstract disagreement sets from τ and ω having variable α;
Let the abstract substitution elements for α in all the abstract disagreement sets be $\alpha/\tau_1, ..., \alpha/\tau_m$ (m > 0);
if α does not occur in τ_i (1 $\leq$ i $\leq$ m) with the exception of recursive type expressions **then**
begin
$\alpha/\tau := \alpha/\tau_1 \oplus ... \oplus \alpha/\tau_m$; $\Sigma := \Sigma \odot < \alpha/\tau >$;
$\tau := \tau \circ < \alpha/\tau >$; $\omega := \omega \circ < \alpha/\tau >$
end
end
else α occurs in τ_i. Fail and return $\Sigma := \emptyset$ and $\delta := \phi$
end { while };
$\delta := \tau := \omega$. Return Σ and δ
end.

Figure 2: Algorithm for Abstract Unification

If the abstract unification of two abstract terms τ and ω gives the *mgcat* as δ and the corresponding ab-

stract substitution Σ, then $\tau o \Sigma$, $\omega o \Sigma$, and δ are abstract equivalent, but not necessarily syntactically identical. The abstract unification of abstract equivalent abstract terms gives the abstract substitution ϵ, the identity abstract substitution. For example, the abstract term $a(\alpha, \{\mathcal{Z}, f(\mathcal{C})\})$ abstract unifies with $a(\mathcal{Z}, \{\mathcal{C}, g(\mathcal{Z})\})$ to give $\Sigma = < \alpha/\mathcal{Z} >$ and *mgcat* $a(\mathcal{Z}, \mathcal{Z})$.

3.3 Summarization

Summarization is the computation of the *least upper bound* for the type expressions associated with the type variables in the calling mode of the abstract goal after the abstract interpretation of the abstract goal. Summarization is used to compute the abstract bindings for the success mode of the calling abstract goal. Summarization is also commutative.

For example, the summarization of $\mathcal{C}$ and $\mathcal{Z}$ gives $\mathcal{C}$. The summarization of $\{f(\mathcal{Z}), \mathcal{Z}\}$, $\{g(\mathcal{Z}), \mathcal{C}\}$ gives $\{f(\mathcal{Z}), g(\mathcal{Z}), \mathcal{C}\}$. The summarization of the indeterminate type μ and any type expression τ gives τ. The summarization of the null type ϕ and any type expression τ gives ϕ.

The algorithm for summarization is given in Figure 3.

Algorithm summarization ;
Input: Abstract substitution elements α/τ, α/ω;
Output: Abstract substitution element α/σ;
begin
if τ (or ω) = μ **or** ϕ **then** $\sigma := \omega$ (or τ)
elseif τ (or ω) includes ω (or τ) **then** $\sigma := \tau$ (or ω)
elseif τ (or ω) is non recursive and ω (or τ) is a recursive type expression and both are abstract unifiable **then**
$\sigma := \omega$ (or τ)
elseif τ (or ω) and ω (or τ) are recursive type expressions with same common subexpression δ and periodicities m and k*m respectively **then**
τ (or ω) includes ω (or τ). $\sigma := \tau$ (or ω)
elseif τ and ω are not set of type expressions and they do not match **then** $\sigma := \{\tau, \omega\}$
elseif τ is an ordered set $\{\tau_1, ..., \tau_m\}$ and ω is $\{\omega_1, ..., \omega_n\}$ **then**
begin
Initialize S to $\emptyset$;
for each pair (τ_i, ω_j) $(1 \leq i \leq m, 1 \leq n)$ **do**
$\delta :=$ summarization of τ_i and ω_j. $S := S \bigcup \{\delta\}$;
Remove all $t_i \in S$ included in $t_j \in S$, $i \neq j$;
$\sigma := S$
end
end.

Figure 3: Summarizing the abstract substitutions elements

3.4 Concretization

Concretization is the process of associating the program variable in the standard domain to the binding of the corresponding type variable in the abstract domain to derive the type information associated with every variable in the program.

Abstract interpretation of the non recursive clause *append*([ ], X, X) of the procedure *append/3*, given in Section 4.2, gives initial mode *append*(nil, $\mathcal{Z}^*$, $\mathcal{Z}^*$) and the final mode *append*(nil, $\mathcal{Z}^*$, $\mathcal{Z}^*$). After concretization the abstract binding associated with the program variable X is $\mathcal{Z}^*$.

4 Putting the Pieces Together

4.1 Basic Interpreter

Given a query Q and a program P, Q and P are *generalized* to give Q^G, the calling mode of Q and P', the generalized program. At each resolution step, G^G, the calling mode of the abstract goal G abstract unifies with the head of an abstract clause $A = H$:- B_1, ..., B_n with abstract substitution Θ. G^G is reduced to a conjunction of abstract subgoals B_1, ..., B_n. Θ forms the initial binding environment E_0 of A. At any point, the abstract instance of the abstract subgoal $B_{i+1} \circ E_i$ forms B^G_{i+1}, the calling mode of the abstract goal B_{i+1}. The abstract subgoal B_{i+1} is abstract interpreted using B^G_{i+1}. The result is B^S_{i+1}, the success mode of the abstract subgoal B_{i+1}. The abstract substitution $\Theta^S_{B_{i+1}}$ formed during abstract unification of B^G_{i+1} and B^S_{i+1} is used to update the binding environment E_i to get E_{i+1}.

The new binding environment E_{i+1} is formed by first creating E'_i from E_i by copying the abstract bindings for all the variables in E_i and then performing the following three operations: **(1)** The abstract unification of the abstract bindings of the common type variables both in E'_i and $\Theta^S_{B_{i+1}}$. **(2)** The inclusion of the abstract bindings for those type variables which occur in E'_i but not in $\Theta^S_{B_{i+1}}$. **(3)** The inclusion of the binding for those type variables which occur in $\Theta^S_{B_{i+1}}$ but not in E'_i.

After the abstract interpretation of the last abstract subgoal B_n, the final binding environment E_n is used to find $A^I \circ E_n$, the *final mode of the clause*, and is represented as A^F. The final mode A^F is abstract unified with B^G_{i+1} to give G^{IS}, the individual success mode due to the abstract interpretation of A. Finally, all these individual success modes from all the clauses with the same relation name and arity are *summarized* to give the success mode G^S for the abstract goal G. After the abstract interpretation is over, the program is *concretized*. A formal algorithm is given in (Bansal 1988).

4.2 Recursive Data Structures

Recursive data structures in logic programs are constructed top down or bottom up. In top down construction, the goals in the body incrementally instantiate the subparts of the structure created in the clause head.

For structures built bottom up, the output variable is bound to a recursive type expression (accumulators are a specific case) at the terminating condition. These two cases, which can be distinguished syntactically, need to be treated separately by abstract interpretation.

Before giving an example of each, we discuss our representation of type expressions that are built during abstract interpretation. We associate a directed graph with each type expression. The graph contains three basic node types: functor nodes, union nodes, and leaf nodes. The leaf nodes can either be a basic type or a type variable.

Recursive type expressions are handled by including a fourth node type, recursion nodes. Recursion nodes are leaf nodes representing recursion point in a data structure. A list of integers, $\mathcal{Z}^*$, is represented by the type graph given in Figure 4. The recursion node L3 is marked as $rec(\alpha)$. The expression $\alpha/\{nil,[\mathcal{Z} \mid rec(\alpha)]\}$ is the textual equivalent of the type graph.

An algorithm for computing the type graph for recursive data structures is given in (Bansal 1988). It depends on the knowledge of recursive procedures. Recursive procedures are determined with a variation of the standard algorithm for computing the strongly connected components of a directed graph, with clause heads as nodes and subgoals as edges, which takes time linear in the total number of clauses and goals. We summarize some salient details here. Recursion nodes are marked in order to distinguish them from void variables and for termination of the abstract unification algorithm when two recursive type expressions are not abstract unifiable. The position of recursion nodes is determined by indexing the leaf nodes with type variables. The indices for branches of all the functor nodes are marked in left to right and ascending order. Union nodes do not affect the indices as they represent alternatives. For example, in the type graph in Figure 4, the recursion node L3 $rec(\alpha)$ has index [2] and the node L1 has index [ ].

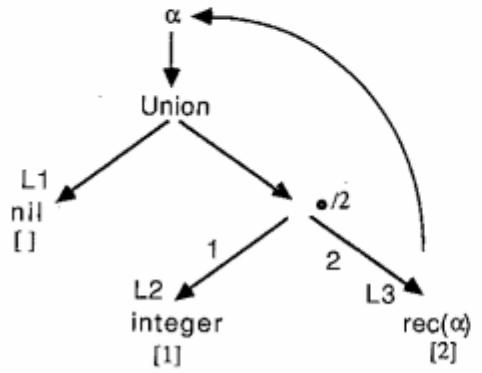

Figure 4: Graph Representation of $\mathcal{Z}^*$

During abstract interpretation, each time an abstract goal G abstract unifies with an abstract clause head of a recursive clause, and the abstract clause is a member of the set of its ancestors, then a recursion cycle is complete. The intermediate success mode G^S is set to be G^C. After the abstract interpretation of all the clauses defining G, summarization is done. At this point, recursion nodes are identified and recursive type expressions are formed. G is abstract interpreted until the last two success modes match.

We note that our procedure does not distinguish between self-recursive procedures and mutually recursive sets of procedures. All that is necessary is the structure of recursion cycles in the program.

The success and calling modes of the abstract subgoals, and the initial and final modes of the clauses invoked after the invocation of the first recursive clause, are updated only after the calling and success mode information is available for the top level goal invoking the first recursive clause.

We now give an example of abstract interpretation for both a top down and bottom up construction of a recursive data structure. The first involves the prototypical top down program, *append*.

```
append([ ], βs, βs).
append([α | αs], βs,[α | γs]) :-
    append(αs, βs, γs).
```

Let the calling mode of the abstract goal *append/3* be $append(\alpha s, \beta s, \mathcal{Z}^*)$.

The type variables αs and βs are unbound. The initial mode of the nonrecursive clause after the abstract unification is $append(\text{nil}, \mathcal{Z}^*, \mathcal{Z}^*)$. The final mode of the nonrecursive clause is same as the initial mode. Similarly, the initial mode of the recursive clause is $append([\mathcal{Z} \mid \alpha s], \beta s,[\mathcal{Z} \mid \mathcal{Z}^*])$ with the initial binding environment E_0 as $< \gamma s/\mathcal{Z}^* >$. The index for the type variable αs is [2] and for βs is [ ]. After the detection of the recursion cycle, summarization of the abstract bindings for the top level goal *append/3* gives the abstract binding for αs as $\{ nil, [\mathcal{Z} \mid \alpha_1 s] \}$ and βs of the form $\beta_1 s$. The index [2] of αs is a proper prefix of the index [2, 2] of the variables $\alpha_1 s$. Therefore αs is a recursion node and it is marked. However, index [ ] for $\beta_1 s$ is same as βs. Therefore, βs is a void variable not involved in the formation of the recursive data structure. The final mode of the recursive clause becomes $append([\mathcal{Z} \mid \mathcal{Z}^*], \mathcal{Z}^*, \mathcal{Z}^*)$. After summarization of the bindings returned by both the clauses the success mode of the top level goal *append/3* becomes $append([\mathcal{Z} \mid \mathcal{Z}^*], \mathcal{Z}^*, \mathcal{Z}^*)$.

We now give an example of bottom up building of recursive data structures. The type variable β associated with recursive data structure (such as accumulator) is initially bound to a base element and keeps building. The output variables are instantiated by uni-

fication with β at the termination of recursion. For example, take the recursive abstract predicate *reverse/3* as described below

reverse(αs, βs) :- *reverse*(αs, *nil*, βs).
reverse(*nil*, γ, γ).
reverse([α | αs], γ, βs) :-
    reverse(αs, [α | γ], βs).

the type graph for the accumulator γ is built bottom up from *nil*. For the calling mode *reverse*($\mathcal{Z}^*$, γs), the calling mode of the abstract subgoal *reverse/3* is *reverse*($\mathcal{Z}^*$, *nil*, γs). After abstract unification with nonrecursive abstract clause *reverse/3*, the initial mode for the abstract clause is *reverse*(*nil, nil, nil*) which also becomes the final mode. The initial mode for the recursive abstract clause of the predicate *reverse/3* is *reverse*([$\mathcal{Z}$ | $\mathcal{Z}^*$], *nil*, γs). The second invocation of the predicate *reverse/3* gives initial mode of the nonrecursive clause *reverse*(*nil*, [$\mathcal{Z}$], [$\mathcal{Z}$]). The type expression [$\mathcal{Z}$] is transformed to represent recursive type expression $\mathcal{Z}^*$. The final mode of the nonrecursive clause is *reverse*(*nil*, $\mathcal{Z}^*$, $\mathcal{Z}^*$). The initial mode of the recursive clause is *reverse*(αs, [$\mathcal{Z}$], γs). The type expression [$\mathcal{Z}$] is transformed to give the recursive type expression $\mathcal{Z}^*$. The intermediate initial mode is also the intermediate final mode for the recursive clause. Summarization gives the success mode of the abstract subgoal *reverse/3* as *reverse*($\mathcal{Z}^*$, *nil*, $\mathcal{Z}^*$).

4.3 Built-in Predicates and Negation

The calling mode information for the built-in deterministic predicates $>$, $<$, $\leq$, $\geq$, *add/3*, and *diff/3* is predefined. If the calling mode of the abstract goal matches with built-in initial mode of the predicates then corresponding final mode is returned. For the built-in predicates, such as *functor/3*, there can be more than one mode; arguments may be polymorphic type expressions. For example, the mode check for the initial mode of *functor/3* is

functor_mode(α, β, γ) :-
    equal(γ, $\mathcal{Z}$), *unbound*(β),
    bound(α), *functor*(α,_,_).

The corresponding final mode is *functor*(α, $\mathcal{C}$, $\mathcal{Z}$).

The negated goal *not G* does not provide any new abstract binding. and the binding environment of the abstract clause remains unaltered after the abstract interpretation. However, the abstract interpretation of G is done normally as if negation were absent.

5 Related Work

A history of the use of abstract interpretation in logic programming is given by Sondergaard (1987). The recent interest in abstract interpretation was sparked by the work of Mellish (1986). He does not take into account the equivalence class of variables formed by unification causing aliasing problem as demonstrated by Debray (1986). The aliasing problem is not an issue for our abstract interpreter because of the more detailed abstract domain.

Since last year, Bruynooghe et. al. (1987 and 1988) have published papers proposing the use of type expressions in abstract interpretation. Recently, Bruynooghe and Jenssens (1988) have proposed an integration of mode and type information, similar to our scheme. However, their scheme does not take into account **(1)** Inclusion polymorphism and recursive type expressions fully **(2)**Information of the equivalence class formed by unification of the two unbound type variables. In their scheme, aliasing problem is removed by repeating the goal until instantiation state does not alter any more. This scheme is computationally inefficient and less formal.

Zobel has suggested a scheme to derive polymorphic type information (Zobel 1987). His analysis is bottom up. Although his type unification scheme takes into account unification of unbound type variables, it does not handle *inclusion polymorphism* and recursive type expressions fully. It is not clear how his type unification algorithm will terminate while unifying two recursive type expressions such as *even* $\mathcal{Z}^*$ and *odd* $\mathcal{Z}^*$. His scheme does not find a fixpoint in the case of mutually recursive predicates and may lead to incomplete type information.

Our scheme is top down in contrast to (Zobel 1987). We derive mode information (in terms of polymorphic type expressions). Our abstract unification scheme also takes into account inclusion polymorphism and equivalence class formed by abstract unification of the type variables and solving the aliasing problem naturally. Mode analysis is done at the same time as abstract interpretation. Our scheme has been applied to detect the producer-consumer relationship. Producers are uninstantiated in the calling mode and instantiated in the success mode. Similarly, consumers are instantiated in the calling mode. The type expression in modes preserves the instantiation information. We use this mode information for a new classification of procedures incorporating integration of classification based on *number of solutions* and *determinacy* (Bansal 1988). Our abstract interpretation scheme can also be used for type generation in a logic program.

6 Conclusions

We have given a scheme for the abstract interpretation of logic programs which integrates type expressions with mode information. We have developed a formal scheme

for abstract unification which incorporates both parametric polymorphism, inclusion polymorphism, and equivalence class of the unbound type variables. The problem of aliasing is not present.

This abstract interpretation can be used efficiently for type generation, producer consumer relationship, and classification of procedure for efficient compilation of logic programs to committed-choice AND-parallelism.

Acknowledgements

This work was supported by the Center of Automation and Intelligent Systems Research at Case Western research University through its core research program funded by CAMP through the State of Ohio.

References

[Bansal 1988] Bansal, A. K., *Incorporating Parallelism In Logic Programs using Program Transformation*, Ph. D. Thesis, Department of Computer Science, Case Western Reserve University, Cleveland, OH 44106, USA, July 1988.

[Bansal and Sterling 1987] Bansal, A. K., and Sterling L., *On Source-to-Source Transfromation of Sequential Logic Programs to AND-parallelism, Proc. of the International Conference on Parallel Processing*, August 1987, St. Charles, Illinois, USA, pp. 795-802.

[Bansal and Sterling 1988] Bansal, A. K., and Sterling, L. S., *Compiling Enumerate-and- Filter Programs for Efficient Execution under Committed-choice AND-parallelism, Proc. of the International Conference on Parallel Processing*, August 1988, St. Charles, Illinois, USA, pp. 22-26.

[Bruynooghe et. al. 1987] Bruynooghe, M., Jenssens, G., Callebaut, A., and Demoen, B., *Abstract Interpretation: Towards the Global Optimization of PROLOG Programs*, Proc. 4th Symposium on Logic Programming, San Francisco, USA, Sept. 1987, pp. 192-204.

[Bruynooghe and Jenssens 1988] Bruynooghe, M. and Jenssens, G., *An Instance of Abstract Interpretation Integrating Type and Mode Inferencing*, Proceedings 5th International Conference of Logic Programming, Seattle, USA, 1988, pp. 669-683.

[Cardelli and Wegner] Cardelli, L., and Wegner, P., *On Understanding types, data abstraction and polymorphism*, ACM Computing Surveys, Volume 17, Number 4, Dec. 1985, pp. 471-522.

[Chang et. al. 1985] Chang, J. H., Despain, A. M., and Degroot, D.,*AND-parallelism of Logic programs based on Static Data Dependency Analysis*, Digest of papers of COMPCON, Spring 1985, pp. 218-225.

[Debray 1986] Debray,S. K., *Automatic Mode-inference for Prolog Programs*, Proc. of the International Symposium of Logic Programming, Salt Lake City, Utah, USA, 1986, pp. 78-88.

[Lloyd 1984] Lloyd, J. W., *Foundations of Logic Programming*, Springer-verlag, New York, 1984.

[Mellish 1981] Mellish, C. S., *An Automatic generation of Mode declarations for Prolog Programs*, DAI Research paper 163, Department of Artificial Intelligence, University of Edinburugh, U. K., August 1981.

[Mellish 1986] Mellish, C. S., *Abstract Interpretation of Prolog Programs*, Proceedings Third International Conference on Logic Programming, London, U. K., July 1986.

[Milner 1978] Milner, R., *A Theory of Polymorphism in Programming*, Journal of Computer and System Sciences, Volume 17, Dec. 1978, pp. 348-375.

[Mishra and Reddy 1985] Mishra, P., and Reddy, U. S., *Declaration free Type Checking*, Conference Record of The Twelth ACM Symposium on Priniciples of Programming Languages, New Orleans, Louisiana, Jan. 1985, pp. 7-21.

[Mycroft and O'Keefe 1984] Mycroft, A., and O'Keefe, R. A., *A Polymorphic Type System for Prolog*, Artificial Intelligence 23, 1984, pp. 295-307.

[Shapiro 1987] Shapiro, E., *Concurrent Prolog-Collected Papers*, Editor E. Shapiro, Volume I and II, MIT Press, Cambridge, Massachussets, 1987.

[Sondergaard 1987] Sondergaard, H., *Abstract Interpretation of Logic Programs*, DIKU Project 86-7-10, University of Copenhagen, January 1987.

[Sterling and Shapiro] Sterling, L., and Shapiro, E.,*The Art of Prolog*, MIT Press, 1986.

[Zobel 1987] Zobel, J., *Derivation of Polymorphic Types for Prolog Programs*, Proc. 4th International Conference of Logic Programming, Melbourne, Australia, June 1987, pp. 817-838.