Chapter 1

Semantic Types for Logic Programs

by Nevin Heintze and Joxan Jaffar

A semantic type represents an approximation to the meaning of a logic program. This paper deals with the relationship between two main formalisms used to define such types. In the first, a set formula is constructed from the program and a type is a distinguished model of this formula. In the second, an adaptation of the immediate consequence operator T_P is defined, and a type is some fixpoint of this operator. In this paper, we use variations of these formalisms to obtain a number of different definitions of type. Our main result is that the formalisms are essentially equivalent: a definition of type in one formalism has its counterpart in the other formalism.

1.1 Background

A type is a formalism for describing an approximation of certain aspects of a program. In logic programming, one can distinguish between two kinds of type: that which pertains to the program's operational model, and that which pertains to the program's declarative semantics. We characterize the first kind as those which describe properties in terms of program derivations. We characterize the second kind, call them the *semantic types*, as the types which describe properties in terms of *successful* program derivations. We now elaborate.

In the former kind of type system, the essential property that is captured is that a derivation step from a well-typed subgoal (that is, a subgoal consistent with the type at hand) can only result in another subgoal which is also well-typed. An example of such a system is [10] where, starting with a definition of type as an annotation of the alphabet, rules were established for constructing well-typed programs and goals. It was then shown that for any such program, type consistency is preserved by a derivation step. A different approach is based directly on program derivations, rather than program syntax. Starting with a distinguished goal, flow analysis or abstract interpretation techniques may be employed to obtain a conservative approximation of the set of possible subgoals that may arise; see [7], [1] for example. Thus, by construction, this approximation has the property that the goals it describes are closed under derivation steps.

A semantic type describes an approximation to the program's meaning. Typically, this approximation is a conservative approximation to a programs success set. That is, it describes a superset of the (ground) atoms which may succeed. An important property captured by such a type is that any atom which falls outside the type is guaranteed not to have a successful derivation. A central issue with semantic types is the trade-off between two requirements: that a type be a recursive structure and that a type give an accurate approximation.

In the literature, three main approaches have been used to obtain semantic types. The first approach obtains from a program a collection of formulas which describes set-theoretic relationships between the atoms appearing in the program. A type is then defined to be a distinguished model of the formulas. The paper [9] introduced this approach, and gave partial algorithms for extracting a type from the formulas. The second approach uses an adaptation of the definition of the immediate consequence operator T_P associated with the program P. This adaptation typically introduces an approximation which ignores argument and/or variable value dependencies. A type is then defined as a fixpoint (or a pre-fixpoint or post-fixpoint) of the resulting function. [14] is an example. The third approach uses flow analysis to infer a semantic type from a program. The algorithms employed are typically based on an abstraction of the underlying domain and an appropriate adaptation of a logic programming operational model. Some examples are [6], [13] and [1].

The relative merits of these three approaches are essentially the following. The first approach presents the type in a closed-form; thus the problem of reasoning about a type can be solved by a decision procedure for the corresponding formulas. The second approach is more declarative in the sense that relationships between the type and the program's meaning (for example, that the approximation given by the type is indeed conservative) are easier to establish. The third approach has lead to practical algorithms based on iterative fixpoint computations. However, to ensure that these algorithms terminate, the abstract domains employed must be restricted; this in turn imposes a fundamental limitation on accuracy.

The primary concern of this paper is the relationship between the set formula and immediate consequence operator approaches for semantic types[1]. We do not

[1] We do not consider (parametric) polymorphism. Although polymorphic type systems have been developed using variations of the set formula approach [11], a corresponding approach based on an immediate consequence operator appears to be problematic.

consider the flow analysis approach essentially because the exact properties of the types inferred are usually dependent on algorithmic details. Thus these types are difficult to characterize in any way other than their algorithmic definition. Consequently, they are generally incomparable with the types from the fixpoint or set formula approaches. In many cases, however, flow analysis approaches employ, in addition to an abstraction of the underlying domain, an approximation which ignores dependencies between arguments. The motivation for this is usually efficiency. In such cases, the set formula and fixpoint approaches can be viewed as an upper bound on the accuracy of the flow analysis approach.

In this paper we consider the class of definite logic programs. We first discuss a uniform methodology for obtaining, given a program, a collection of set formulas representing the desired type. From here, we obtain a number of different ways of obtaining such formulas which, in turn, give rise to a number of different types. These types coincide with those appearing in the literature. We then present an alternative framework for obtaining a number of different adaptations of T_P. The main theorem proves that the various notions of type obtained via set formulas are respectively equivalent to the various notions of type obtained via adaptations of T_P. Formal proofs of the various results are somewhat lengthy and are omitted.

1.2 Preliminaries

We refer to [8] for standard terminology and definitions about pure logic programs P. We assume, with no loss of generality, that no two rules in P have the same variable. Given P, we write Π_P and Σ_P to respectively denote the finite sets of predicate and function symbols in P. Let HU_P and HB_P respectively denote the Herbrand universe and base arising from Π_P and Σ_P. Let $P|_p$ denote the rules in P defining the relation p. Where $S \subseteq HB_P$, let $S|_p$ denote the subset of S involving the predicate p. A *body-only* (*head-only*) variable within a rule is one which does not appear in the head (body) of the rule.

Where e is any expression, $var(e)$ denotes its collection of variables. An atom or term is *linear* if it contains at most one occurrence of each distinct variable. The symbol $\sim$ is used to group expressions into a possibly empty sequence or set, this choice being dictated by an obvious context. For example $A \leftarrow \tilde{B}$ denotes a rule R whose body atoms are represented by $\tilde{B}$. Writing $\tilde{B} \subseteq S$, on the other hand, means that each atom in $\tilde{B}$ appears in the set S.

A *set expression* (s.e.) over an alphabet Δ of functors is either

- a variable in VAR;
- a constant from Δ or one of the special constants $\bot$ or $\top$;
- $f(t_1, \ldots, t_n)$ where $f \in \Delta$ is n-ary, $n \geq 1$, and $t_1, \ldots, t_n$ are s.e.'s

- $f^{-1}_{(i)}(s)$ where $f \in \Delta$ is n-ary, $1 \leq i \leq n$, and s is a s.e.
- $\bar{s}$ where s is a s.e.;
- $u \uplus s$, $u \cup s$, or $u \cap s$ where u and s are s.e.'s.

A *set constraint* is of the form $s \subseteq t$ where s and t are set expressions.

Let $\tau(\Delta)$ denote the set of all (ground) terms over Δ. Let $S, S_1, \cdots, S_n$ be subsets of $\tau(\Delta)$ and let f be an n-ary functor, $n \geq 1$. Then $f(S_1, \cdots, S_n)$ denotes the set

$$\{f(s_1, \cdots s_n) : s_1 \in S_1, \cdots, s_n \in S_n\}.$$

Define the ith *projection* of f, call it $f^{-1}_{(i)}$, by

$$f^{-1}_{(i)}(S) \stackrel{\text{def}}{=} \{s_i : f(s_1, \cdots, s_n) \in S\}.$$

The following defines a *tuple closure* operation $\star$ on sets of ground terms:

$$S^\star \stackrel{\text{def}}{=} \{c : c \text{ is a constant in } S\} \cup \bigcup_f f\left((f^{-1}_{(1)}(S))^\star, \cdots, (f^{-1}_{(arity(f))}(S))^\star\right)$$

where f ranges over the symbols whose arity, denoted $arity(f)$, is non-zero.

An *interpretation* $\mathcal{I}$ is a mapping from each variable in VAR into a subset of $\tau(\Delta)$. An interpretation can be uniquely extended to map from set expressions to subsets of $\tau(\Delta)$ as follows. If t is:

- a variable, then $\mathcal{I}(t)$ is already defined;
- a constant symbol c, then $\mathcal{I}(t)$ is $\{c\}$;
- a special constant $\perp$ or $\top$, then $\mathcal{I}(t)$ is $\emptyset$ or $\tau(\Delta)$ respectively;
- $f(t_1, \ldots, t_n)$ where $n \geq 1$, then $\mathcal{I}(t)$ is $f(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$;
- $f^{-1}_{(i)}(s)$, then $\mathcal{I}(t)$ is $f^{-1}_{(i)}(\mathcal{I}(s))$;
- $\bar{s}$, then $\mathcal{I}(t)$ is $\tau(\Delta) - \mathcal{I}(s)$;
- $u \uplus s$ or $u \cup s$, then $\mathcal{I}(t)$ is $(\mathcal{I}(u) \cup \mathcal{I}(s))^\star$ or $\mathcal{I}(u) \cup \mathcal{I}(s)$ respectively;
- $u \cap s$, then $\mathcal{I}(t)$ is $\mathcal{I}(u) \cap \mathcal{I}(s)$.

A *model* of a collection $\mathcal{C}$ of set constraints is an interpretation of $\mathcal{C}$ which evaluates to true. Where $\mathcal{I}_1$ and $\mathcal{I}_2$ are interpretations, $\mathcal{I}_1 \subseteq \mathcal{I}_2$ denotes that $\mathcal{I}_1(X) \subseteq \mathcal{I}_2(X)$ for all variables X. We use $lm(\mathcal{C})$ and $gm(\mathcal{C})$ to respectively denote the least and greatest models of $\mathcal{C}$ (ordered according to $\subseteq$).

1.3 Set Formulas

The use of set formulas to model program behavior dates back to at least [12] which infers data structure definitions for a lisp-like language. In logic programming, set formulas were used in [9] to obtain an approximation of the success set of a program. We begin this section with an outline of the method for constructing such formulas corresponding to a program. We then introduce refinements to this basic method.

The set formulas constructed below are intended to describe the success set of a program P. To begin, we introduce a distinguished set of variables called *type variables* which have the form Υ_p where p is a predicate symbol in P. By construction, any model of these formulas will map each type variable Υ_p into a set of ground atoms. Consequently, a model of the set formulas defines an interpretation of a program P as the union of the sets corresponding to the type variables, that is:

$$\bigcup_{p \in \Pi_P} \mathcal{I}(\Upsilon_p).$$

In what follows, we will sometimes identify a model of set formulas with a subset of HB_P in the sense just described. We now proceed with the construction of a set formula corresponding to P.

First obtain, for each predicate symbol p in Π_P, a *head-formula* of the form

$$\Upsilon_p \subseteq H_{p,1} \uplus H_{p,2} \uplus \cdots \uplus H_{p,n_p}$$

where $H_{p,1}, \cdots, H_{p,n_p}$ denote the head-atoms in $P|_p$. Then obtain, for each body atom $q(\tilde{t})$ in P, a *body-formula* of the form

$$q(\tilde{t}) \subseteq \Upsilon_q.$$

An interpretation and model for such set formulas is essentially the same as those given for set constraints in the previous section. Note, however, that each type variable is assigned a subset of HB_P while each program variable is assigned a subset of HU_P. Consider an example program P:

```
p(a).
p(f(X)) :- p(X).
```

The set formula constructed from P is

$$\begin{array}{rcl} \Upsilon_p & \subseteq & p(a) \uplus p(f(X)) \\ p(X) & \subseteq & \Upsilon_p. \end{array}$$

The greatest model of the above formula gives $\Upsilon_p = \{p(f^n(a)) : n \geq 0\}$, and happens to capture exactly the meaning of the program. The least model, on the other hand, is the trivial model which maps all variables into $\emptyset$. (This is true in general for the formulas constructed above; the least model of these formulas is therefore not useful for defining program approximations).

The formulas in [9] were essentially constructed as indicated above, and the distinguished model sought was the *greatest* model. This choice is not always appropriate, as the following example shows. Consider augmenting the above program with the rule `p(Z) :- p(Z)`. The corresponding set formula now becomes

$$\begin{array}{rcl} \Upsilon_p & \subseteq & p(a) \uplus p(f(X)) \uplus p(Z) \\ p(X) & \subseteq & \Upsilon_p \\ p(Z) & \subseteq & \Upsilon_p. \end{array}$$

While the meaning the program has not changed with the addition of the extra rule, the greatest model of the set formula now assigns HB_P to Υ_p. That is, the greatest model gives no useful information about the program.

We now present a refinement to the scheme above by considering a change not in the choice of model, but in the set formulas themselves. Specifically, we replace set inclusion by set equality and appropriately modify the right-hand-sides of the body-formulas. Let P be a program with $N \geq 1$ rules. First define that $HEADONLY_P$ is the set of equations

$$\{Z = \top : Z \text{ is a head-only variable in } P\}$$

and define that $HEAD_P(p)$, is

$$\Upsilon_p = H_{p,1} \uplus \cdots \uplus H_{p,n_p}$$

where $H_{p,1}, \cdots, H_{p,n_p}$ denote the head atoms in $P|_p$[2]. Now, for any $1 \leq i \leq N$, let the variables in the i^{th} rule be X_1, X_2, $\cdots$, X_k where $k \geq 0$, and let the body of this rule be the sequence of atoms $p_1(\tilde{t_1})$, $\cdots$, $p_n(\tilde{t_n})$ where $n \geq 0$. Then, where $x_1, \cdots, x_k$ is a sequence of distinct new variables, obtain a new sequence $p_1(\tilde{t_1}')$, $\cdots$, $p_n(\tilde{t_n}')$ by replacing each variable X_i by the corresponding new variable x_i. Now define that $BODY_P(i)$, $1 \leq i \leq N$, is the conjunction of the following equations:

$$X_j = \left\{ x_j : \exists x_1 \cdots x_{j-1} x_{j+1} \cdots x_k \left(\begin{array}{rcl} p_1(\tilde{t_1}') & \in & \Upsilon_{p_1} \\ p_2(\tilde{t_2}') & \in & \Upsilon_{p_2} \\ & \vdots & \\ p_n(\tilde{t_n}') & \in & \Upsilon_{p_n} \end{array} \right) \right\}, \quad 1 \leq j \leq k.$$

[2] When $P|_p$ is the empty set, $HEAD_P(p)$ is defined to be $\Upsilon_p = \perp$.

We are now in a position to define that the set formula SF_P corresponding to a given program P is the conjunction of the following equations:

$$HEADONLY_P \quad \cup \quad \left(\bigcup_{p \in \Pi_P} HEAD_P(p) \right) \quad \cup \quad \left(\bigcup_{1 \leq i \leq N} BODY_P(i) \right).$$

Using the previous example program P

```
p(a).
p(f(X)) :- p(X).
p(Z) :- p(Z).
```

the set formula SF_P is

$$\begin{array}{rcl} \Upsilon_p & = & p(a) \uplus p(f(X)) \uplus p(Z), \\ X & = & \{x : p(x) \in \Upsilon_p\}, \\ Z & = & \{z : p(z) \in \Upsilon_p\}. \end{array}$$

The definition of an interpretation and model for such set formulas is obtained by extending the definition given previously in an obvious way. Note, however, that the symbol $\top$ is always interpreted as HU_P.

Given any program P, it can be proved that *all* models of SF_P define conservative approximations to the meaning of P. It is therefore appropriate to choose the least model $lm(SF_P)$ (which always exists) as our distinguished model. In the above example, $lm(SF_P)$ describes the exact meaning of P, $\{p(f^n(a)) : n \geq 0\}$. We remark that the greatest models of SF_P correspond to the greatest models of the formulas used in [9] which are described in the first part of this section.

A variation to the construction of SF_P described above is obtained by replacing all occurrences of the symbol $\uplus$ by the symbol $\cup$. To distinguish between these two variations of SF_P, we now introduce the notations $SF_P(\uplus)$ and $SF_P(\cup)$. One immediate observation is that the least model of $SF_P(\cup)$ is a better approximation to the meaning of P than $SF_P(\uplus)$. Consider, for example, the program consisting of just the two facts `p(a, b)` and `p(b, a)`. Here $SF_P(\uplus)$ and $SF_P(\cup)$ would respectively be

$$\Upsilon_p = p(a,b) \uplus p(b,a) \quad \text{and} \quad \Upsilon_p = p(a,b) \cup p(b,a),$$

giving the least models

$$\Upsilon_p = \{p(a,b), p(b,a), p(a,a), p(b,b)\} \quad \text{and} \quad \Upsilon_p = \{p(a,b), p(b,a)\}$$

respectively. We now define yet another variation on the construction methods for $SF_P(\uplus)$ and $SF_P(\cup)$. We proceed first by example: let P be the program

```
p(a, b).
p(b, a).
q(X) :- p(X, X).
```

The new set formula is:

$$\begin{array}{rcl} \Upsilon_p & = & p(a,b) \cup p(b,a), \\ \Upsilon_q & = & q(X), \\ X & = & \{x : \exists x' p(x,x') \in \Upsilon_p\} \cap \{x : \exists x' p(x',x) \in \Upsilon_p\}. \end{array}$$

Consider the difference between this formula and $SF_P(\cup)$, that is, consider the third equation above. In $SF_P(\cup)$, this equation appears as $X = \{x : p(x,x) \in \Upsilon_p\}$, whereas in the new formula, a set is constructed for each *occurrence* of the variable X. The final set for X is then determined by the intersection of the values corresponding to the various occurrences. Note that the least model of the formula above gives $\Upsilon_q = \{q(a), q(b)\}$, whereas the least model of $SF_P(\cup)$ gives $\Upsilon_q = \emptyset$.

We now extend our notation $SF_P(\uplus)$ and $SF_P(\cup)$ to $SF_P(\uplus, var)$, $SF_P(\uplus, occ)$, $SF_P(\cup, var)$ and $SF_P(\cup, occ)$ to distinguish (a) the use of $\uplus$ or $\cup$ in the head-formulas, and (b) the treatment of each variable as one entity or in an occurrence-based way. We now formally define $SF_P(\uplus, occ)$; the definition of $SF_P(\cup, occ)$ will then be obtained by simply replacing each occurrence of $\uplus$ by $\cup$.

Fix i: $1 \leq i \leq N$. Let the i^{th} rule of P be $H \leftarrow p_1(\tilde{t_1}), \cdots, p_n(\tilde{t_n})$ where $n \geq 0$ and let $\tilde{B}$ denote $p_1(\tilde{t_1}), \cdots, p_n(\tilde{t_n})$. Fix on some variable X appearing in $\tilde{B}$, and let m denote the number of occurrences of X in $\tilde{B}$. Let the remaining variables in $\tilde{B}$ be $Y_1, \cdots, Y_k$ and let $\tilde{y} = y_1, \cdots, y_k$ be a corresponding sequence of distinct new variables. Then, obtain a new sequence $p_1(\tilde{t_1}'), \cdots, p_n(\tilde{t_n}')$ by replacing each occurrence of a variable in $p_1(\tilde{t_1}), \cdots, p_n(\tilde{t_n})$ as follows:

1. replace the jth occurrence of X, $1 \leq j \leq m$, by a new variable x_j, and

2. replace every occurrence of Y_l, $1 \leq l \leq k$, by the corresponding variable y_l.

Now define $BODY_P^{occ}(i)|_X$ to be the equation:

$$X = \begin{array}{ll} & \left\{ x_1 : \exists x_2 x_3 \cdots x_m \exists \tilde{y} \begin{pmatrix} p_1(\tilde{t_1}') & \in & \Upsilon_{p_1} \\ & \vdots & \\ p_n(\tilde{t_n}') & \in & \Upsilon_{p_n} \end{pmatrix} \right\} \\ \cap & \left\{ x_2 : \exists x_1 x_3 \cdots x_m \exists \tilde{y} \begin{pmatrix} p_1(\tilde{t_1}') & \in & \Upsilon_{p_1} \\ & \vdots & \\ p_n(\tilde{t_n}') & \in & \Upsilon_{p_n} \end{pmatrix} \right\} \\ \cap & \cdots \\ \cap & \left\{ x_m : \exists x_1 \cdots x_{m-1} \exists \tilde{y} \begin{pmatrix} p_1(\tilde{t_1}') & \in & \Upsilon_{p_1} \\ & \vdots & \\ p_n(\tilde{t_n}') & \in & \Upsilon_{p_n} \end{pmatrix} \right\} \end{array}$$

Next define that $BODY_P^{occ}(i)$ is the conjunction of the equations $BODY_P^{occ}(i)|_X$ over the variables X in $\tilde{B}$. Finally, define that $SF_P(\uplus, occ)$ is the conjunction of the following equations:

$$HEADONLY_P \quad \cup \quad \left(\bigcup_{p \in \Pi_P} HEAD_P(p) \right) \quad \cup \quad \left(\bigcup_{1 \leq i \leq N} BODY_P^{occ}(i) \right)$$

where $HEADONLY_P$ and $HEAD_P(p)$ are as defined before.

In summary for this section, we have defined four different ways to construct, given a logic program P, a set formula whose least model represents the type of P. The resulting four classes of set formulas: $SF_P(\uplus, var)$, $SF_P(\uplus, occ)$, $SF_P(\cup, var)$ and $SF_P(\cup, occ)$, differed only in whether or not set-union was augmented with a tuple-closure operation, and in whether or not program variables were treated in an occurrence-based way. Informally, these formulas are intended to be natural closed-form representations of conservative approximations to the program's meaning. Formally, we have

Lemma 1 *For any logic program P,*

$$\begin{aligned} T_P{\uparrow}\omega \ &\subseteq \ lm(\mathrm{SF}_P(\cup, var)) \\ &\subseteq \ lm(\mathrm{SF}_P(\cup, occ)) \\ &\subseteq \ lm(\mathrm{SF}_P(\uplus, var)) \\ &= \ lm(\mathrm{SF}_P(\uplus, occ)). \quad \Box \end{aligned}$$

1.4 Fixpoint Operators

We give three different adaptations of the immediate consequence function T_P associated with a program P. These, in turn, define three different kinds of types. We begin by recalling the standard T_P operator:

$$T_P(\mathcal{I}) \quad \stackrel{\text{def}}{=} \quad \left\{ H\theta : H \leftarrow \tilde{B} \in P, \text{ and } \tilde{B}\theta \subseteq \mathcal{I} \right\}.$$

Observe that this definition consists of three main components: (i) the collection of unifiers (for example, the collection of θ's above) corresponding to the body atoms of a rule, (ii) the application of these unifiers to the head of a rule, and (iii) the joining of the resulting sets, one from each rule head. The adaptations of T_P considered in this section shall be obtained by replacing some of these components by approximate counterparts.

First consider the fixpoint operator defined in [14]. This function, call it Y_P, is defined to be the tuple closure of the T_P function. That is:

$$Y_P(\mathcal{I}) \quad \stackrel{\text{def}}{=} \quad (T_P(\mathcal{I}))^{\star}.$$

Thus components (i) and (ii) of the definition of T_P remain unchanged in Y_P. Component (iii), the joining of head atoms, is now augmented with a tuple closure operation. For example, if P were

```
p(f(a, b)).
p(f(b, a)).
r(X) :- p(f(X, X)).
s(f(Y, Z)) :- p(f(Y, Z)).
```

then $T_P{\uparrow}\omega$ is $\{p(f(a,b)),\ p(f(b,a)),\ s(f(a,b)),\ s(f(b,a))\}$, whereas $Y_P{\uparrow}\omega$ is the larger set

$$\left\{\begin{array}{l} p(f(a,a)),\ p(f(a,b)),\ p(f(b,a)),\ p(f(b,b)) \\ r(a),\ r(b) \\ s(f(a,a)),\ s(f(a,b)),\ s(f(b,a)),\ s(f(b,b)) \end{array}\right\}.$$

Note that in the construction of $Y_P{\uparrow}\omega$ the approximation operator $\star$ is applied after each application of T_P. Thus, for example, it is generally not the case that $Y_P{\uparrow}\omega = (T_P{\uparrow}\omega)^{\star}$.

Recently, in [4], a more accurate approximate consequence operator was studied. This operator, call it $\mathcal{T}_P$, was obtained by first defining an approximation operator $\mathcal{A}$ which is applied to a collection of unifiers to obtain a *set substitution*. Such a substitution is a mapping from terms into sets of ground terms, and it can be defined as the natural extension of a mapping from variables into sets of ground terms. Thus, for example, if α were the set substitution given by the mapping $\{[X \mapsto \{a,c\}, Y \mapsto \{b,d\}]\}$, then $p(X,Y)\alpha$ denotes the set $\{p(a,b),\ p(a,d),\ p(c,b),\ p(c,d)\}$.

Now let Θ be a collection of ground substitutions over variables $\tilde{X}$. Define that $\mathcal{A}_{\tilde{X}}(\Theta)$ is the single set substitution which maps each variable $X_i \in \tilde{X}$ into the set $\{X_i\theta : \theta \in \Theta\}$, and it is undefined on all other variables[3]. The operator $\mathcal{T}_P$ can now be defined as follows:

$$\mathcal{T}_P(\mathcal{I}) \stackrel{\text{def}}{=} \left\{ a \in H\alpha : \begin{array}{l} H \longleftarrow \tilde{B} \in P, \text{ and} \\ \alpha = \mathcal{A}_{var(H \leftarrow \tilde{B})}(\{\theta : \tilde{B}\theta \subseteq \mathcal{I}\}) \end{array} \right\}.$$

Intuitively, $\mathcal{T}_P$ first collects together the ground substitutions for a rule which instantiate the body atoms into elements in $\mathcal{I}$. From these substitutions, it collects all the possible values that each variable may be instantiated to, ignoring relationships between these variables. A set substitution is then defined as the mapping from each variable into the corresponding collected set of values, and finally, this set substitution is applied to the head of the rule.

[3] $\mathcal{A}_{\tilde{X}}(\emptyset)$ is a special set-substitution which maps every term into $\emptyset$. If however $\Theta \neq \emptyset$ and $\tilde{X}$ is empty, $\mathcal{A}_{\tilde{X}}(\Theta)$ is the set-substitution which maps every ground term t into $\{t\}$.

Note that the type of closure used here is different from the tuple closure operation used to define Y_P. In Y_P, the closure occurred at the level of arguments of predicate and function symbols so that, for example, tuple closure augments the set $\{p(a,b), p(c,d)\}$ with the elements $p(a,d)$ and $p(c,b)$. In $\mathcal{T}_P$, closure occurs at the level of *variables*, so that, the unifiers $\{[X \mapsto a, Y \mapsto b], [X \mapsto c, Y \mapsto d]\}$, for example, are approximated by the set substitution $\{[X \mapsto \{a,c\}, Y \mapsto \{b,d\}]\}$.

Returning to the previous example program P, $\mathcal{T}_P{\uparrow}\omega$ is

$$\left\{ \begin{array}{l} p(f(a,b)), p(f(b,a)) \\ s(f(a,a)), s(f(a,b)), s(f(b,a)), s(f(b,b)) \end{array} \right\}.$$

An alternative definition of an approximate consequence operator is obtained by using substitutions which are defined to be mappings from each *occurrence* of a variable, as opposed to mappings from each variable. Toward this end, define a *occurrence-based substitution* to be of the form

$$\left\{ X_{j_1}^{(i_1)}/t_1, X_{j_2}^{(i_2)}/t_2, \cdots\ X_{j_n}^{(i_n)}/t_n \right\}$$

where $n \geq 1$, the $t_1, \cdots, t_n$ are terms, $\tilde{X} = X_{j_1}, \cdots X_{j_n}$ is a sequence of not necessarily distinct variables, and $\{i_1, \cdots, i_n\}$ is a collection of positive numbers such that i_k identifies a specific occurrence, the i_k^{th} one, of the variable X_{j_k}. Application of a occurrence-based substitution on a sequence of terms or atoms is defined in the obvious way. For example $p(X), q(X,Y)\{X^{(1)}/a, X^{(2)}/b, X^{(3)}/c, Y^{(1)}/d, Y^{(2)}/e\}$ is $p(a), q(b,d)$. An occurrence-based substitution θ is *ground on* a sequence of terms or atoms $\tilde{D}$ if θ grounds all variable occurrences in $occ(\tilde{D})$, the set of variable occurrences in $\tilde{D}$. We can now adapt the $\mathcal{A}$ approximation operator defined above so that it maps collections of ground occurrence-based substitutions with respect to $\tilde{D}$ as follows: $\mathcal{A}_{occ(\tilde{D})}(\Theta)$ maps each variable X in $var(\tilde{D})$ into the set:

$$\left\{ \bigcap_{X^{(i)} \in occ(\tilde{D})} \{X^{(i)}\theta : \theta \in \Theta\} \right\}$$

and is undefined if X does not appear in $\tilde{D}$. We now define yet another operator Z_P in a similar way as $\mathcal{T}_P$, except that occurrence-based substitutions θ are now used instead of normal substitutions, as follows:

$$Z_P(\mathcal{I}) \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} a : a \in H\alpha & \text{where} & H \longleftarrow \tilde{B} \in P, \text{ and} \\ & & \alpha = \mathcal{A}_{occ(\tilde{B},H)}(\{\theta : \tilde{B}\theta \subseteq \mathcal{I}\}) \end{array} \right\}$$

where θ ranges over the ground occurrence-based substitutions on the sequence $\tilde{B}, H$. Using the above program P once again, $Z_P{\uparrow}\omega$ is

$$\left\{\begin{array}{l} p(f(a,b)), p(f(b,a)) \\ r(a), r(b) \\ s(f(a,a)), s(f(a,b)), s(f(b,a)), s(f(b,b)) \end{array}\right\}.$$

The relative accuracies of the three operators indicated by this example program are true in general. That is, Z_P gives rise to an approximate meaning of a program which is, in general, more accurate than Y_P but less accurate than $\mathcal{T}_P$. The following lemma is easy to verify.

Lemma 2 *For any logic program P,*

$$\begin{array}{rcl} T_P{\uparrow}\omega & \subseteq & \mathcal{T}_P{\uparrow}\omega \\ & \subseteq & Z_P{\uparrow}\omega \\ & \subseteq & Y_P{\uparrow}\omega. \quad \square \end{array}$$

1.5 Main Results

We now prove that the formulation of types via set formulas is essentially equivalent to the formulation of types using fixpoint operators. Specifically, the desired result is that for any program P,

- a model of $SF_P(\uplus, var)$ is a fixpoint of Y_P, and vice versa;
- a model of $SF_P(\cup, occ)$ is a fixpoint of Z_P, and vice versa;
- a model of $SF_P(\cup, var)$ is a fixpoint of $\mathcal{T}_P$, and vice versa.

However, the simple program `p(a) :- p(b).` is a counter-example. A closer observation reveals that the falsity of the desired result can be isolated to programs which have very specific properties[4]. We deal with these special properties now.

Let $\mathcal{I}$ be a Herbrand interpretation. A rule R is *redundant w.r.t.* $\mathcal{I}$ if $T_{\{R\}}(\mathcal{I}) = \emptyset$. A program P is *redundant* if it contains a rule which is redundant w.r.t. $T_P{\uparrow}\omega$. A rule is *degenerate* if its head atom is ground. A program is *degenerate* if it contains a degenerate rule.

Theorem 1 *Let P be a program which is* <u>either</u> *non-redundant or non-degenerate, and let $\mathcal{I} \subseteq \mathrm{HB}_P$. Then*

- *$\mathcal{I}$ models $\mathrm{SF}_P(\cup, var)$ iff $\mathcal{I}$ is a fixpoint of $\mathcal{T}_P$;*
- *$\mathcal{I}$ models $\mathrm{SF}_P(\cup, occ)$ iff $\mathcal{I}$ is a fixpoint of Z_P;*

[4] We could modify the definitions of the various set formulas so that their models correspond to fixpoints over arbitrary programs. This would, however, make the construction of a set formula corresponding to a program slightly more complex.

- $\mathcal{I}$ *models* $\mathrm{SF}_P(\uplus, var)$ *iff* $\mathcal{I}$ *is a fixpoint of* Y_P;
- $\mathcal{I}$ *models* $\mathrm{SF}_P(\uplus, occ)$ *iff* $\mathcal{I}$ *is a fixpoint of* Y_P. $\square$

One important application of this theorem is that fixpoint types can be computed using algorithms for the corresponding set formulas. Even though the theorem is stated conditionally in terms degeneracy and redundancy, it is in fact applicable for all programs. Specifically, suppose that an algorithm computes a representation of $SF(\cup, var)$ given an *arbitrary* program P. We assume that one can determine whether or not a sequence of atoms $\tilde{B}$ has a ground instance in the representation. We can now apply the above theorem showing how to obtain a representation of $\mathcal{T}_P\!\uparrow\!\omega$.

If P is non-degenerate (and this is an easily tested syntactic condition), then the theorem implies that the representation we have is in fact the representation we seek. If P were degenerate, then we can apply the following algorithm.

> *Set P_1 to the subset of P containing just the non-degenerate rules. Compute a representation of* $\mathrm{SF}_{P_1}(\cup, var)$. *Now, for each degenerate rule in $P - P_1$, determine if its body atoms have a ground instance in the representation. If so, add the rule to P_1. Let P_2 denote the resultant set of rules, and repeat this process to obtain P_3, P_4, etc. In other words, to compute P_i, $i \geq 2$, use a representation of* $\mathrm{SF}_{P_{i-1}}(\cup, var)$, *and construct P_i from P_{i-1} by adding those rules in P which have a ground instance in* $\mathrm{SF}_{P_{i-1}}(\cup, var)$. *Clearly the process can be terminated when P_i is the same as P_{i-1} for some i, and the resulting representation is indeed that of $\mathcal{T}_P\!\uparrow\!\omega$.*

An alternative approach is to rewrite each rule R in P as follows. Let Z be a new variable, then add an extra argument, filled with the variable Z, to each atom in R. For example `p(a) :- p(b).` would be rewritten to `p(a, Z) :- p(b, Z).` The resulting program, call it P', is non-degenerate, and, by systematically ignoring the extra argument, it can be shown that P' is essentially equivalent to P with respect to fixpoint and set formula constructions. Therefore, to compute a representation of $SF_P(\cup, var)$, it suffices to compute a representation of $SF_{P'}(\cup, var)$, and then delete the extra argument from this representation.

In summary, our theorem proves that the formulation of types via set formulas is equivalent to the formulation of types using using fixpoint operators in the context of programs P which were either non-degenerate or non-redundant. The theorem, however, is essentially applicable over arbitrary programs because it is straightforward to augment any algorithm which computes a representation of the least model of a given set formula so that it computes a representation of the least fixpoint of the corresponding fixpoint operator.

1.6 Concluding Discussion

We have established an equivalence between two formalisms for defining semantic types, and three specific types were studied. An additional way to study the relative properties of these three types is obtained by program transformation. Given a program P, two kinds of transformation are possible. We briefly describe their properties here, but omit the full details.

- We may obtain a program P' such that $\mathcal{Z}_{P'}{\uparrow}\omega$ and $\mathcal{T}_{P'}{\uparrow}\omega$ coincide and such that $\mathcal{Z}$ is preserved, that is, $\mathcal{Z}_P{\uparrow}\omega = \mathcal{Z}_{P'}{\uparrow}\omega = \mathcal{T}_{P'}{\uparrow}\omega$. This transformation is essentially the exhaustive application of the replacement steps such as: if $H \leftarrow A, p(X,X), B$ is a rule, then replace this rule by $H \leftarrow A, p(X, X_1), p(X_2, X), B$ where X_1 and X_2 are new variables. Note also that $\mathcal{SF}_P(\cup, occ)$ is equivalent to $\mathcal{SF}_{P'}(\cup, var)$, and so this program transformation can be used to obtain an alternative definition for $\mathcal{SF}_P(\cup, occ)$.
- We may obtain a program P' such that $\mathcal{T}_{P'}{\uparrow}\omega$ and $T_{P'}{\uparrow}\omega$ coincide and such that $\mathcal{T}$ is preserved, that is, $\mathcal{T}_P{\uparrow}\omega = \mathcal{T}_{P'}{\uparrow}\omega = T_{P'}{\uparrow}\omega$. This transformation is essentially the exhaustive application of the replacement steps such as: if $q(X,Y) \leftarrow p(X,Y)$ is a rule, then replace this rule by $p(X,Y) \leftarrow p(X, Z_1), p(Z_2, Y)$, where Z_1 and Z_2 are new variables.

Transformations are also possible on set formulas. In fact, it is possible to transform both $\mathcal{SF}(\cup, occ)$ and $\mathcal{SF}(\uplus, var)$ so that they become just set constraints (that is, it is possible to eliminate the quantifiers). This process, called "flattening", is useful for two reasons. First, it leads to constraints of a simpler and more intuitive structure. Second, it reduces the problem of reasoning about set formulas to the simpler problem of reasoning about set constraints.

We now conclude with a brief summary of the three kinds of semantic type we have described.

- $\mathcal{T}_P{\uparrow}\omega$ or $lm(\mathcal{SF}_P(\cup, var))$:
  This is the most accurate type of the three. It has been shown to be a recursive set [4], and further, there are algorithms to compute a representation of the type in the form of a regular tree grammar (in the sense of nondeterministic tree automata; see e.g. [2]). Also, it is possible to transform P to a program P' such that $\mathcal{T}_P{\uparrow}\omega = T_{P'}{\uparrow}\omega$. However, it is not in general possible to apply flattening to the set formula $\mathcal{SF}_P(\cup, var)$ to reduce it to set constraints.
- $\mathcal{Z}_P{\uparrow}\omega$ or $lm(\mathcal{SF}_P(\cup, occ))$:
  This is the type of intermediate accuracy. That these types are recursive can be shown using transformations on either the program, or the corresponding set formulas. First, it is possible to transform P to a program P' such that $\mathcal{Z}_P{\uparrow}\omega = \mathcal{T}_{P'}{\uparrow}\omega$. Hence we can use the algorithm for $\mathcal{T}_P{\uparrow}\omega$. Alternatively, the

flattening procedure can be applied to $SF_P(\cup, occ)$ to reduce these set formulas to set constraints, and these constraints can be solved by an algorithm from [5]. Thus these types can also be represented by regular tree grammars.

- $Y_P\uparrow\omega$ or $lm(SF_P(\uplus, var))$ or $lm(SF_P(\uplus, occ))$:
  This is the least accurate type of the three. The question of whether this type is recursive is still open. Although it is possible to apply the flattening procedure to $SF_P(\uplus, var)$ to reduce these set formulas to set constraints, algorithms do not exist for the resulting constraints. It is conjectured that these types can be represented by the subclass of regular tree grammars corresponding to deterministic root-to-frontier tree automata. This type has, however, been shown to useful in a type-checking system [14].

Finally, we remark that the definitions of set formulas may be weakened so as to obtain different kinds of types. Specifically, we can replace $=$ with $\supseteq$ or $\subseteq$ in the head equations, to obtain post- and pre-fixpoints of the corresponding operators. An earlier manuscript [3] used this variation to define types for notions such as specializing a program's meaning, and run-time type consistency.

1.7 Acknowledgements

We thank Michael Maher for his comments.

Bibliography

[1] M. Bruynooghe, G. Janssens, A. Callebaut and B. Demoen, "Abstract Interpretation: Towards the Global Optimisation of PROLOG Programs", *Proceedings* 4^{th} *IEEE Symposium on Logic Programming*, pp 192 - 204, September 1987.

[2] F. Gècseg and M. Steinby, "Tree Automata", Akadèmiai Kiadò, Budapest, 1984.

[3] "Elementary Logic Programming Types", draft manuscript, July 1987.

[4] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", IBM Research Report RC 16089 (#71415), 6 Sept. 90. (An extended abstract appears in *Proc. ACM-POPL-90.*)

[5] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Set Constraints", forthcoming CMU techical report, February 1991. (An extended abstract appears in *Proc. IEEE LICS-90.*)

[6] T. Kanamori and K. Horiuchi, "Type Inference in PROLOG and its Applications", *Proc.* 9^{th} *Intl. Joint Conf. on Artificial Intelligence*, Vol. 2, pp 704 - 707, 1985.

[7] F. Kluzniak, "Type Synthesis for Ground PROLOG", *Proceedings* 4^{th} *Intl. Conf. on Logic Programming*, MIT Press, pp 788 - 816, July 1987.

[8] J.W. Lloyd, "Foundations of Logic Programming", Second Edition, Springer-Verlag Series on Symbolic Computation, 1987.

[9] P. Mishra, "Toward a Theory of Types in PROLOG", *Proceedings* 1^{st} *IEEE Symposium on Logic Programming*, pp 289 - 298, 1984.

[10] A. Mycroft and R.A. O'Keefe, "A Polymorphic Type System for PROLOG", *Artificial Intelligence* 23, pp 295 - 307, 1985.

[11] C. Pyo and U.S. Reddy, "Inference of Polymorphic Types for Logic Programs" *Logic Programming, Proceedings of the North American Conference, 1989*, MIT Press, pp 1115 - 1134, October, 1989.

[12] J.C. Reynolds, "Automatic Computation of Data Set Definitions", *Information Processing 68*, North-Holland, pp 456 - 461, 1969.

[13] J. Xu and D.S. Warren, "A Type Inference System for PROLOG", *Proceedings* 5^{th} *Intl. Conf. and Symp. on Logic Programming*, MIT Press, pp 604 - 619. July 1988.

[14] E. Yardeni and E.Y. Shapiro, "A Type System for Logic Programs", in *Concurrent PROLOG: Collected Papers*, Vol. 2, MIT Press, pp 211 - 244, 1987.