Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A THEORY OF DIRECTED LOGIC PROGRAMS AND STREAMS

D. Stott Parker
R. R. Muntz

April 1988
CSD-880031

A Theory of
Directed Logic Programs and Streams †

D. Stott Parker

R.R. Muntz

Computer Science Department
University of California
Los Angeles, CA 90024-1596

ABSTRACT

For some time it has been recognized that logic programmers commonly write *directed predicates,* i.e., predicates supporting only certain input and output patterns among their arguments. In many logic programming implementations, programmers are encouraged to use 'mode declarations' to announce this directedness, both as a matter of style and as a directive for compiler optimization.

A common application of directed programming is stream or list processing. Programs that operate on streams or lists usually have specific input and output arguments. More generally, directed predicates can represent functions, with specific inputs and outputs.

We present a new declarative formalism for directedness in logic programming systems. The formalism is based on the use of partial ordering constraints rather than unification. Semantics of the resulting system are rigorously definable, and extend ordinary logic program semantics in a natural way.

The approach to directed logic programs presented here will probably provide higher performance than is possible with undirected programs. Furthermore, the approach provides perspective relating diverse concepts such as predicate 'modes', functional computation, constraint processing, and stream processing.

† This work was done within the **Tangram** project, supported by DARPA contract F29601-87-C-0072.

A Theory of
Directed Logic Programs and Streams †

D. Stott Parker
R.R. Muntz
Computer Science Department
University of California
Los Angeles, CA 90024-1596

1. Introduction

Tangram is a Prolog-based environment for modeling being developed at UCLA [16]. Its emphasis is on processing both non-numeric and numeric models, in a framework that combines management of complex structures (e.g., models) and regular flat structures (e.g., relational databases containing large quantities of model output). We feel that most of the parallelism to be exploited in modeling and in real non-numeric programs lies in stream processing. Consequently, one of our main goals is the introduction of efficient stream processing into Prolog. This goal is clearly shared by the designers of AND-parallel Prolog systems. A simple, efficient mechanism for stream communication is vital.

There have been many attempts to incorporate stream processing into logic programming systems such as Prolog. For a number of reasons, this incorporation is not easy. Among other things, Prolog does not naturally support clause access other than with backtracking [23], and does not support multiple processes working on streams. Also, many performance tradeoffs arise in reconciling the power of logic programming with the realities of practical stream processing.

Unification, or some analogue of unification, is often relied upon to provide a communication mechanism in systems supporting stream processing. This is a natural approach, since unification forms the basis of communication in ordinary logic programs. Unfortunately, unification does not properly capture a notion of 'flow', i.e., directed communication, that is common in stream processing. Unification is not directed; it is a two-way affair. Capturing the notion of flow correctly is important not only for conceptual elegance and correctness (e.g., in dealing with communication of partially instantiated terms [14] – terms containing variables) but also for performance, since it affects both control and communication overhead. Two-way communication with unification is inherently more complex and expensive than the one-way communication found in ordinary streams.

As a result, the treatment of unification differs among parallel Prolog systems. For example, let us consider a simple PARLOG merge program [9] taking two input streams (the

first and second arguments) and producing an output stream (the third argument). A *mode declaration* is used to indicate this directedness, i.e., that the first two arguments are expected as input, and the third is output:

```
mode Merge(?,?,^).
Merge([x|a],b, [x|c]) <- Merge(a,b,c).
Merge(a,[x|b], [x|c]) <- Merge(a,b,c).
Merge([],b,b).
Merge(a,[],a).
```

This program can be put in Parlog's *standard form* by using *one-way unification* against all input arguments:

```
mode Merge(?,?,^).
Merge(v1,v2,v3) <- [x|a] <= v1, b <= v2      : Merge(a,b,c), v3 = [x|c].
Merge(v1,v2,v3) <- a <= v1,     [x|b] <= v2  : Merge(a,b,c), v3 = [x|c].
Merge(v1,v2,v3) <- [] <= v1,    b <= v2      : v3 = b.
Merge(v1,v2,v3) <- a <= v1,     [] <= v2     : v3 = a.
```

The one-way unification primitive (x <= y) binds variables only in x so as to make x and y unify, if this is possible. Briefly:

(1) If there is a way to bind variables in x to make it unify with y without also binding variables in y, then these bindings are made and the call to <= succeeds;

(2) otherwise, if there is a way to unify x and y, then the call to <= suspends;

(3) otherwise the call to <= fails.

Thus, one-way unification is more a pattern-matching operation than unification. One-way unification can be used to reduce the overhead of full unification, but at a price in complexity of semantics. Simpler mechanisms for communication would be helpful here.

It turns out that the one-way unification operator above is very similar to $\sqsubseteq$, the partial ordering of generality (subsumption) among terms. That is, (x <= y) is very close semantically to the logical constraint ($x \sqsubseteq y$), which requires the term x to be *less general than* the term y, i.e., that x must be *subsumed* by y. For example, given the constraint $(f(x,y) \sqsubseteq f(a,g(b)))$, we must have $x = a$ and $y = g(b)$.

When x and y are terms that have no variables in common, $x \sqsubseteq y$ is essentially identical semantically to x <= y. In this case $x \sqsubseteq y$ is 'one-way'. That is, x is constrained by y, but not vice versa. A simple implementation of this case in Prolog could be as follows, using Edinburgh Prolog syntax with X and Y instead of x and y:

```
X⊑Y  :- copy_term(Y,T),  unify(X,T).
```

Our thesis is that logic programs can use $\sqsubseteq$ as a basic primitive. It would be a mistake to discard this connection between <= and $\sqsubseteq$ as an interesting curiousity. In many situations $\sqsubseteq$ can take the place of unification. In particular, use of ordering constraints like $\sqsubseteq$ is natural in stream processing. For example, we will study stream-processing logic programs like the *merge* program below:

$$
\begin{array}{llllll}
merge(V1,V2,V3) & \leftarrow & [X \mid A] \sqsubseteq V1, & B \sqsubseteq V2, & [X \mid C] \sqsupseteq V3, & merge(B,A,C). \\
merge(V1,V2,V3) & \leftarrow & A \sqsubseteq V1, & [X \mid B] \sqsubseteq V2, & [X \mid C] \sqsupseteq V3, & merge(B,A,C). \\
merge(V1,V2,V3) & \leftarrow & [\,] \sqsubseteq V1, & B \sqsubseteq V2, & B \sqsupseteq V3. & \\
merge(V1,V2,V3) & \leftarrow & A \sqsubseteq V1, & [\,] \sqsubseteq V2, & A \sqsupseteq V3. &
\end{array}
$$

More generally, we feel that replacement of unification by a variety of *ordering constraints* provides an elegant way to define ordering relationships among loosely-coupled computations, such as those in stream processing. Here $\sqsubseteq$ is an important ordering, but many others can be used.

This approach gives a declarative way to deal with the more general issue of *directed predicates,* predicates supporting only certain input and output patterns among their arguments. Programs that operate on streams typically have specific input arguments and output arguments, defining a directed 'flow' of stream processing. Directedness manifests itself whenever predicates represent functions; some of the arguments represent inputs to the function, while others represents the function's output.

The perspective offered by this approach gives us an interesting handle on stream-based computation in general. Many of the basic ideas in this paper appear in Gregory's book [9], but the formal approach here raises and clarifies important issues. The approach also helps in appreciating generalizations of streams, such as Tribble et al's *channel* [26], a stream-like object providing partial ordering among data items. This approach also makes formal connections between stream processing and constraint programming, as represented by Prolog II [7], Constraint Logic Programming (CLP) [10, 11], and Partial Order Programming [22].

2. Foundations

In this section we review basic results for partial orders and the subsumption ordering on first-order terms. In the interest of clarity and a self-contained presentation we reproduce a number of standard definitions. We then go on to explore properties of term subsumption constraints.

2.1. Partial Orders

Definition

A *partial order* is a pair $<D,\sqsubseteq>$ where $\sqsubseteq$ is a binary relation on D such that

(P1) For all x in D, $x \sqsubseteq x$.
(P2) For all x,y,z in D, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$.
(P3) For all x,y in D, if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.

A *preorder* is a pair $<D,\sqsubseteq>$ satisfying (P1) and (P2).

A *total order* $<D,\sqsubseteq>$ is a partial order in which for every pair of elements x,y either $x \sqsubseteq y$ or $y \sqsubseteq x$ holds.

Definition

In a partial order $<D,\sqsubseteq>$, an *upper bound* of a subset S of D is an element z in D such that for every x in S, $x \sqsubseteq z$. The *least upper bound* of a set S, written $\bigsqcup S$, is the least z such that z is an upper bound of S. By P3, it is unique if it exists.

Definition

A *directed set* S in a partial order $<D,\sqsubseteq>$ is a nonempty subset of D with the property that if x,y are arbitrary elements in S, then there is another element z in S such that both $x \sqsubseteq z$ and $y \sqsubseteq z$.

A *complete partial order* (*cpo*) is a partial order $<D,\sqsubseteq>$ in which every directed subset S of D has a least upper bound $\bigsqcup S$.

Definition

A function on a partial order $f : D \rightarrow D$ is *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

Definition

A function $f : D \rightarrow D$ on a cpo $<D,\sqsubseteq>$ is called *continuous* if for every directed subset S of D,

$$f(\bigsqcup S) = \bigsqcup f(S) .$$

Fixed Point Theorem

Let $f : D \rightarrow D$ be a continuous map on the cpo $<D,\sqsubseteq>$ with a least element $\perp$. Then f has a least fixed point equal to

$$\mathbf{fix}\, f = \bigsqcup \{ f^k(\perp) \mid k \in \omega \}$$

where ω is the natural numbers, f^k is f iterated k times, i.e., $f^k = f \circ f^{k-1}$, and f^0 is the identity. See e.g. [12, 22].

Definition
A *complete lattice* is a partial order $<D, \sqsupseteq>$ such that *every* subset S of D has a least upper bound $\sqcup S$. In particular $\perp \in D$, since $\perp = \sqcup \emptyset$.

Complete lattices are sometimes defined as partial orders in which $\sqcup S$ and $\sqcap S$ exist for every set S. Because $\sqcap S = \sqcup \{ x \mid y \sqsupseteq x, \text{for every } y \text{ in } S \}$, the definition here is equivalent.

2.2. First-Order Terms and Unification

We include here basic material about first-order terms and unification from [15]. An excellent reference is [13], which reviews previous work and clarifies many confusing issues concerning unification.

The set T of *(finite) first-order terms* over a given set of function symbols is defined as follows. Let F be the set of function symbols f of interest, each of which has an arity (number of arguments). Symbols with arity zero are constants. Also, let V be an infinite set of variables. T is the least set defined inductively by the following set of requirements:

(1) Every variable in V is a term in T.

(2) If f is a function symbol of arity n in F, and $t_1, ..., t_n$ are terms in T, then $f(t_1, ..., t_n)$ is a term in T.

For example, if f has arity 2, $F = \{f\}$, and $V = \{X, Y, \cdots\}$, then

$$T = \{ X, Y, f(X,X), f(X,Y), f(Y,X), f(Y,Y), f(X, f(X,X)), \ ... \ \}.$$

For simplicity we will follow the Prolog convention of introducing specific variables with upper-case letters, and specific constants and function symbols with lower-case letters. General terms will be denoted by r, s, t, etc.

A *ground term* is a term that contains no variables.

A *substitution* θ is a member of $V \to T$, i.e., a set of *bindings* $\{X_1 / t_1, \ldots, X_n / t_n\}$ mapping each variable X_i to the first-order term t_i. The variables X_i are distinct, and t_i cannot be X_i.

Applying the binding X_i / t_i to the variable X_i results in the replacement of X_i by the term t_i, and we say X_i is *bound* to t_i.

If t is a term, the *instance* $t\theta$ of t is the result of simultaneous application of all bindings in θ to the corresponding variables in t. A *ground instance* is an instance that is also a ground term.

A substitution θ is a *renaming* if it is of the form $\{X_1 / Y_1, ..., X_n / Y_n\}$ where the X_i are distinct variables, and the Y_i are distinct variables.

Two terms s and t are called *variants* if there exist renamings σ and θ such that both

$s = t\sigma$ and $t = s\theta$. Thus s and t differ only in the names of their variables.

Let $\rho = \{X_1 / r_1, ..., X_m / r_m\}$ and $\sigma = \{Y_1 / s_1, ..., Y_n / s_n\}$ be substitutions. Then the *composition* $\rho\sigma$ is the substitution obtained from the set

$$\{ X_1 / r_1\sigma, ..., X_m / r_m\sigma,\ Y_1 / s_1, ..., Y_n / s_n \}$$

by then deleting any binding $X_i / r_i\sigma$ for which $r_i\sigma$ is X_i, and deleting any binding Y_j / s_j for which Y_j is some X_k for some k. Composition is *associative*, i.e., for all substitutions ρ, σ, θ, and all terms t, $\rho(\sigma\theta) = (\rho\sigma)\theta$, and $t(\sigma\theta) = (t\sigma)\theta$.

A substitution θ is *idempotent* if $\theta\theta = \theta$.

A *unifier* of two terms s and t is a substitution θ such that $s\theta$ and $t\theta$ are identical. A unifier θ is a *most general unifier (mgu)* of s and t if for every other unifier σ, there exists a substitution γ such that $\sigma = \theta\gamma$.

For example, a most general unifier for $s = f(X,b)$ and $t = f(a,Y)$ is $\theta = \{X / a, Y / b\}$, and $s\theta = t\theta = f(a,b)$. This substitution is idempotent.

If s and t have a unifier, they have an idempotent unifier. In fact, Robinson's unification algorithm [15] always produces idempotent most general unifiers.

2.3. First-Order Term Subsumption

First-order terms may be ordered by generality, or subsumption. For example,

$$f(X,Y),\ f(U,f(V,W)),\ f(f(a,Z),Z)$$

are first-order terms, and the first term is called *more general* than either the second or the third. Equivalently we say the first *subsumes* the second and the third. Henceforth we write this relationship of subsumption as a binary relation $\sqsupseteq$ on terms, whose definition includes:

$$f(X,Y) \sqsupseteq f(U,f(V,W)),$$
$$f(X,Y) \sqsupseteq f(f(a,Z),Z).$$

By convention, we will usually write these ordering statements below with $\sqsupseteq$ rather than with $\sqsubseteq$.

Definition

Let s and t be first-order terms.

We say $s \sqsupseteq t$ if there exists an idempotent substitution θ such that $s\theta = t$.

For example, if $s = f(X,Y)$, $t = f(U,f(V,W))$, then $\theta = \{ X / U, Y / f(V,W) \}$ satisfies $s\theta = t$, so $s \sqsupseteq t$. However $f(X,Y) \not\sqsupseteq f(Y,X)$, and $f(X,Y) \not\sqsupseteq f(a,X)$.

Idempotency of θ is included in this definition in order for subsumption to be 'reasonable' when s and t have variables in common. In many situations (such as when analyzing properties of subsumption by itself) subsumption can be defined without insisting on

idempotency of θ, but in the situations we deal with later it is important. For example, with $s = f(X,Y)$, $t = f(Y,X)$, the non-idempotent substitution $\theta = \{ X/Y, Y/X \}$ satisfies $s\theta = t$ (and $t\theta = s$). However, in the context of a logic program it will not be reasonable for $f(X,Y)$ to subsume $f(Y,X)$.

Intuitively, $\sqsupseteq$ is related to $\supseteq$ (set containment). If a term s is more general than t, then the set of instances of s contains the instances of t. Formally, denote by $ground(t)$ the set of all ground instances of t.

Prop. 1
If $s \sqsupseteq t$ then $ground(t) = ground(s) \cap ground(t)$.

The converse of Prop. 1 holds provided that s and t have no common variables, since then it can be proven that $ground(s) \cap ground(t) = ground(t\theta)$, where θ is a most general unifier of s and t. See [27], where $ground(t)$ is referred to as $proj(t)$.

Corollary
If $s \sqsupseteq t$ then $ground(s) \supseteq ground(t)$.

Again the converse holds provided that s and t have no common variables. Note in particular that if $s \sqsupseteq t$ and s is a ground term, then $s = t$.

Prop. 2
If both $s \sqsupseteq t$ and $t \sqsupseteq s$, then s and t are variants.

Definition
A *common instance* of a set of terms S is a term t such that $t \sqsubseteq s$ for every element s in S. The *greatest common instance* $\sqcap S$ is the greatest such instance (modulo variants).
A *common generalization* of a set of terms S is a term t such that $t \sqsupseteq s$ for every element s in S. The *least common generalization* $\sqcup S$ of S is the least such generalization (again modulo variants).

Example
Let $S = \{ f(a,g(Y)), f(X,g(b)), f(a,Z) \}$. Then $\sqcup S = f(X,Z)$, and $\sqcap S = f(a,g(b))$.

Prop. 3
Let T_v be the set of equivalence classes of first-order terms in T, where two terms are equivalent if and only if they are variants. Then $<T_v, \sqsubseteq>$ is a complete lattice.

Proof

To show $<T_v, \sqsubseteq>$ is a partial order, note the equivalence class definition satisfies property P3. The only problem is in showing that property P2 (transitivity) holds. If $s \sqsupseteq t$ and $t \sqsupseteq u$ there are idempotent substitutions σ, θ such that $s\sigma = t$ and $t\theta = u$. Although the composition of idempotent substitutions is not idempotent in general, by applying renaming substitutions to θ if necessary we can force $\sigma\theta$ to be idempotent here, and thus $s \sqsupseteq u$. So $\sqsupseteq$ is transitive.

To show that $<T_v, \sqsubseteq>$ is a complete lattice, we must prove that every set S of term

variants has a least upper bound $\sqcup S$. This follows [24,25] since we can find a least generalization $s \sqcup t$ for any pair of term variants s and t via an *anti-unification* algorithm, and if $S = \{s_1, s_2, \ldots\}$ the sequence $< t_1, t_2, \ldots >$, where $t_1 = s_1$ and $t_i = t_{i-1} \sqcup s_i$ otherwise, must stop increasing in generality at some finite index (giving the least common generalization), since $t_{i-1} \sqsubseteq t_i$ and there cannot be an infinite increasing sequence of finite term variants. □

Prop. 4
$s \sqsupseteq t$ if and only if $t\theta = t$ for some idempotent most general unifier θ of s and t.

Proof

If there is an idempotent mgu θ such that $t\theta = t$, then $s\theta = t\theta = t$, so $s \sqsupseteq t$. Conversely, if $s \sqsupseteq t$ there is an idempotent substitution σ such that $s\sigma = t$. But then $s\sigma\sigma = t\sigma$. However since σ is idempotent, $s\sigma\sigma = s\sigma$, so $t = t\sigma$. Thus σ is an idempotent unifier of s and t. If σ is also a most general unifier, setting $\theta = \sigma$ completes the proof. Otherwise there is an idempotent most general unifier θ and a substitution γ such that $\sigma = \theta\gamma$. We may assume γ includes no renaming substitutions, since we may require θ to include all necessary renamings. But then $t = t\sigma = t\theta\gamma$, so no variable in t is bound by $\theta\gamma$, and hence no variable in t is bound by θ. So $t\theta = t$ also. □

Definition
Let s and t be first-order terms. An *orderer of s over t* is an idempotent substitution θ such that $s\theta \sqsupseteq t\theta$. A *ground orderer of s over t* θ is an orderer that makes $s\theta$ and $t\theta$ ground terms, so $s\theta = t\theta$.

We say θ is a *most general orderer (mgo)* of s over t if for any other orderer σ, there exists a substitution μ such that $\theta\mu = \sigma$.

Prop. 5
There is an orderer of s over t if and only if s and t are unifiable.
Proof

If s and t have an orderer σ, then $s\sigma \sqsupseteq t\sigma$ and $s\sigma\rho = t\sigma$ for some idempotent ρ. But then $s\sigma\rho\rho = t\sigma\rho = t\sigma$, and $\sigma\rho$ is a unifier of s and t. Conversely, if s and t are unifiable, then they have an orderer, since there is a idempotent mgu θ such that $s\theta = t\theta$, so $s\theta \sqsupseteq t\theta$. □

We now show how to construct a most general orderer for unifiable terms.

Definition
Let S be a set of first-order terms. An *impartial substitution* for S is an idempotent substitution $\theta = \{ X_1 / u_1, \ldots, X_n / u_n \}$ where no u_i contains variables of any term in S.

Prop. 6
If two terms s and t are unifiable, they have an impartial most general unifier (that is, an idempotent mgu that is impartial for $\{s, t\}$).
Proof

In any idempotent mgu θ with bindings X / u where u contains Y appearing in s or in t, we can produce a new mgu $\theta' = \theta\rho$ by applying the renaming $\rho = \{Y / V\}$ for some new variable V. Proceeding again in this way we can eliminate all bindings to variables in s or in t. The resulting mgu is also idempotent. □

Theorem 1
Let θ be an impartial mgu for s and t. Define $\theta_{[s,t]} \subseteq \theta$ by

$$\theta_{[s,t]} \; = \; \{ X / u \in \theta \mid X \text{ is a variable that appears in } t \}.$$

Then $\theta_{[s,t]}$ is an mgo of s over t.
Proof

Since θ is idempotent, $\theta = \theta_{[s,t]} \cup (\theta - \theta_{[s,t]}) = \theta_{[s,t]}(\theta - \theta_{[s,t]})$, both $(\theta - \theta_{[s,t]})$ and $\theta_{[s,t]}$ are idempotent, and so $s\theta_{[s,t]} \sqsupseteq s\theta$. But $s\theta = t\theta = t\theta_{[s,t]}$. So $\theta_{[s,t]}$ is an orderer of s over t.

We must show that $\theta_{[s,t]}$ is most general. Let σ be any other orderer of s over t. We claim there is a substitution μ such that $\theta_{[s,t]}\mu = \sigma$.

We know that $s\sigma\rho = t\sigma$ for some idempotent ρ. Thus as in Prop. 5, $s\sigma\rho\rho = t\sigma\rho = t\sigma$, and $\sigma\rho$ is a unifier of s and t. Furthermore, we can assume without loss of generality that ρ introduces renamings if necessary so that no variable in the right of some binding in $\sigma\rho$ also appears in the right of some binding in θ. Since θ is also an mgu, we know there exists a substitution γ such that $\theta\gamma = \sigma\rho$.

Now, let X be any variable appearing either in s or in t, assuming without loss of generality that σ makes bindings only to these variables. If X is in t, then $X\theta_{[s,t]} = X\theta$ and $X\sigma\rho = X\sigma$. Also, since $\theta_{[s,t]}$ contains only bindings X / u where u is a term having no variables that appear only in s,

$$X\theta_{[s,t]}\gamma = X\theta\gamma = X\sigma\rho = X\sigma.$$

On the other hand, if X is not in t, then $X\theta_{[s,t]} = X$, so

$$X\theta_{[s,t]}\sigma = X\sigma.$$

Define μ as follows:

$$\begin{aligned}\mu \; = \; & \{ X / u \in \gamma \mid X \text{ is a variable that appears in } t\theta_{[s,t]} \} \\ \cup \; & \{ Y / v \in \sigma \mid Y \text{ is a variable that appears in } s \text{ but not in } t \}.\end{aligned}$$

Then μ satisfies the claim. □

It may help to look at the situation pictorially. Given the constraint $\leftarrow s \sqsupseteq t$, we must bind variables in s and t appropriately. When θ is an mgu of s and t, $s\theta = t\theta$ is the 'intersection' of s and t. Furthermore s is more general than $s\theta_{[s,t]}$, and $s\theta_{[s,t]}$ is more general than $s\theta$. A portion of the term lattice is shown in Figure 1.

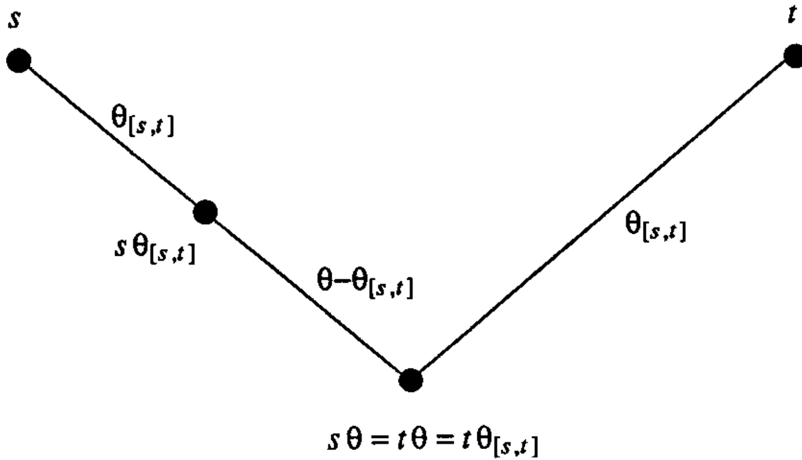

Figure 1. Term Subsumption and $\theta_{[s,t]}$

For example, the terms $s = f(X,Y)$ and $t = f(a,X)$ have most general unifier $\theta = \{ X / a, Y / a \}$, and the subset $\theta_{[s,t]} = \{ X / a \}$ satisfies $s\theta_{[s,t]} \sqsupseteq t\theta_{[s,t]}$. This is shown in Figure 2.

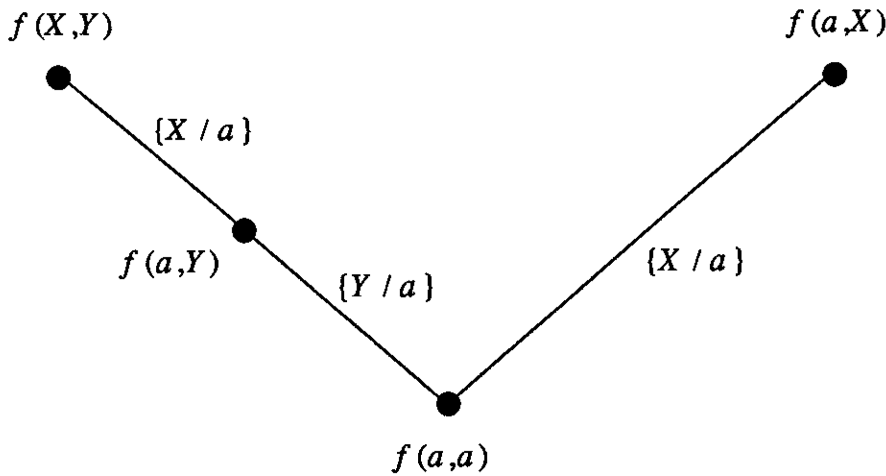

Figure 2. Example of Term Subsumption and $\theta_{[s,t]}$

Theorem 1 shows that $\theta_{[s,t]}$ has the same effect on s and t as a most general unifier of t and a copy of s that retains all the variables s shares with t. In other words, if $s' = s\rho$ where ρ renames all variables in s that are not in t to new variables, then there is a mgu θ' of s' and t such that $t\theta_{[s,t]} = t\theta'$ and $s\theta_{[s,t]} = s\theta'$. For the example above, if we take $\rho = \{ Y / Z \}$, then a mgu of s' and t is $\theta' = \{ X / a, Z / a \}$, and $s\theta' = f(a,Y)$, $t\theta' = f(a,a)$. We will show later how this result can be exploited in implementations of subsumption.

2.4. Term Subsumption Constraints

Let us now investigate how the two-argument predicate $\sqsupseteq$ behaves as a constraint, so that we may include first-order term subsumption in a logic programming system.

Consider first some examples. For the goal

$$\leftarrow a \sqsupseteq Z$$

to be satisfied, Z must be bound to a. Similarly, for

$$\leftarrow f(X,a) \sqsupseteq f(Y,Z)$$

to be satisfied, Z must be bound to a. In general, Theorem 1 tells us that the single logic programming goal

$$\leftarrow s \sqsupseteq t$$

can be satisfied by finding an impartial most general unifier θ for s and t, selecting the subset $\theta_{[s,t]} \subseteq \theta$ of bindings for the variables that appear in t, and then applying $\theta_{[s,t]}$ to the goal. In other words, since $s\,\theta_{[s,t]} \sqsupseteq t\,\theta_{[s,t]}$, the goal and substitution

$$\leftarrow (s \sqsupseteq t)\,\theta_{[s,t]}$$

is satisfied, whether the terms s and t have common variables or not.

In the important case where s and t have no variables in common, $s\,\theta_{[s,t]} = s$, and this goal and substitution are equivalent to

$$\leftarrow s \sqsupseteq (t\,\theta_{[s,t]}).$$

In this case, $s \sqsupseteq t$ is a *one-way constraint.* As long as s and t have no variables in common, only t can be affected by bindings made to satisfy the constraint. This is an important property, as it gives a formal definition for *'one-way unification'* within the existing theoretical framework for logic programming.

It is natural to view $\sqsupseteq$ as a constraint when it appears in a *conjunction* of goals. Above, when we considered the goal

$$\leftarrow f(X,a) \sqsupseteq f(Y,Z)$$

we noticed that Z must be bound to a. Additionally, however, this goal constrains X to subsume Y. That is, all bindings applied to X for other conjuncts must also be applied to Y. For example, with

$$\leftarrow f(X,a) \sqsupseteq f(Y,Z),\ \ X = b$$

Y must be bound to b for the goal to be satisfied. Also, with

$$\leftarrow f(X,W) \sqsupseteq f(Y,Z),\ \ X = W$$

Y must be unified with Z to satisfy the goal. (Why? For $\leftarrow f(X,X) \sqsupseteq f(Y,Z)$ to be satisfied there must be an idempotent θ such that $f(X,X)\theta$ matches $f(Y,Z)$. This is possible only if Y matches Z.)

We call $\sqsupseteq$ a *constraint* because its satisfaction can constrain bindings without actually applying them.

A particularly useful example of a goal conjunction is

$$\leftarrow s \sqsupseteq t,\ t \sqsupseteq s$$

which constrains s and t to be variants, as mentioned in Prop. 2. That is, in order to satisfy both goals any binding applied to s must also be applied to t, and vice versa. This is operationally equivalent to *unifying* s and t. For this reason we use $(s \sqsupseteq t,\ t \sqsupseteq s)$ interchangeably with $(s = t)$ later.

A final interesting example of a goal conjunction is

$$\leftarrow X \sqsupseteq a,\ X \sqsupseteq b.$$

This conjunction is noteworthy since it is satisfied if X remains a variable, yet no assignment of X to a ground term will satisfy it. This example points out that there are *two different domains* over which we can select bindings to satisfy constraints:

(1) the full set T of first-order terms, including variables;
(2) the Herbrand universe of all ground terms.

Definition
A conjunction of one or more constraints G is *Term-satisfiable* if there exists a substitution θ that is an orderer for each constraint in G. G is *Term-satisfied* if each individual constraint in G is satisfied, i.e., the empty substitution is an orderer for each constraint.

A conjunction of constraints G is *Herbrand-satisfiable* if there exists a substitution θ that is a *ground* orderer for each constraint in G. G is *Herbrand-satisfied* if it is Term-satisfied and has a ground orderer.

For example, if G is $(X \sqsupseteq a, X \sqsupseteq b)$, then G is Term-satisfiable (in fact, it is Term-satisfied), but G is not Herbrand-satisfiable.

Definition
A *Term most general orderer (Term mgo)* of a conjunction of subsumption constraints G is a substitution θ such that $G\theta$ is Term-satisfied, and θ is most general: if $G\sigma$ is also Term-satisfied, then there exists γ such that $\theta\gamma = \sigma$.

A *Herbrand most general orderer (Herbrand mgo)* of a conjunction of subsumption constraints G is a substitution θ such that $G\theta$ is Herbrand-satisfied, and θ is most general: if $G\sigma$ is also Herbrand-satisfied, then there exists γ such that $\theta\gamma = \sigma$.

Theorem 2a
Term most general orderers of a conjunction G of subsumption constraints are unique up to renaming, and there is a simple algorithm for computing them.
Proof

Given the conjunction

$$G \ = \ s_1 \sqsupseteq t_1, \ \cdots, s_n \sqsupseteq t_n$$

define

$$g \;=\; [[s_1,t_1]\,,\,\ldots\,,\,[s_n,t_n]]$$

so g is a first-order term. Here [,] is a function symbol that can be used to construct lists. Also define, for $1 \le i \le n$,

$$f_i(\,[[u_1,v_1]\,,\ldots,\,[u_n,v_n]]\,) \;=\; \begin{cases} [[u_1,v_1]\,,\ldots,\,[u_n,v_n]]\;\theta_{[u_i,v_i]} & \text{if } \theta_{[u_i,v_i]} \text{ exists} \\ [[u_1,v_1]\,,\ldots,\,[u_n,v_n]] & \text{otherwise} \end{cases}$$

where $\theta_{[u_i,v_i]}$ is an mgo of u_i over v_i that is impartial for $[[u_1,v_1],\ldots,[u_n,v_n]]$.

We claim that each f_i is monotone, i.e., if $g_1 \dot\sqsupseteq g_2$ then $f_i(g_1) \dot\sqsupseteq f_i(g_2)$. First, if $g_1 \dot\sqsupseteq g_2$ then $g_1\sigma = g_2$ for some idempotent substitution σ, so for each i:

$$\begin{aligned} f_i(g_1) &= g_1\,\theta_{[u_i,v_i]} \\ f_i(g_2) &= g_1\,\sigma\,\theta_{[u_i\sigma,v_i\sigma]}. \end{aligned}$$

Now $\theta_{[u_i\sigma,v_i\sigma]}$ is an mgo of $u_i\sigma$ over $v_i\sigma$, so $\sigma\theta_{[u_i\sigma,v_i\sigma]}$ is an orderer of u_i over v_i. Since $\theta_{[u_i,v_i]}$ is an mgo of u_i over v_i, there is a substitution μ such that

$$\theta_{[u_i,v_i]}\,\mu \;=\; \sigma\,\theta_{[u_i\sigma,v_i\sigma]}.$$

It follows that $f_i(g_1) \dot\sqsupseteq f_i(g_2)$, so f_i is monotone as claimed. Similarly, we can show that each f_i is continuous, i.e., $f_i(\,\sqcap S\,) \;=\; \sqcap f_i(S)$ for any directed set S of terms. This follows from

$$\sqcap S \;=\; \sqcup\{\, t \mid t \,\dot\sqsubseteq s \text{ for all } s \in S \,\} \;=\; \sqcup\{\, s\gamma \mid s \in S,\ \gamma \text{ a substitution } \}.$$

Since finite compositions of continuous functions are continuous, the function

$$F(g) \;=\; f_1(f_2(\ldots f_n(g)\ldots))$$

is continuous. The first-order terms modulo variants form a complete lattice (Prop. 3), so the partial order $<T_v,\sqsubseteq>$ where $\sqsubseteq$ is the *reverse* of $\dot\sqsubseteq$ (i.e., $x \sqsubseteq y$ iff $x \dot\sqsupseteq y$) is a cpo whose least upper bound operator is the greatest common instance operator $\sqcap$ on T_v. Also $g \dot\sqsupseteq F(g)$, so $g \dot\sqsupseteq F(g) \dot\sqsupseteq F^2(g) \dot\sqsupseteq \ldots$ and $g \sqsubseteq F(g) \sqsubseteq F^2(g) \sqsubseteq \ldots$, and as in the Fixed Point Theorem the greatest common instance

$$\sqcap_{k\in\omega} F^k(g)$$

exists and is the least fixed point of F under $\sqsubseteq$ (the greatest fixed point of F under $\dot\sqsubseteq$). Thus we can find an orderer θ for the constraints G, when an orderer exists, by composing the substitutions applied by each f_i in the fixed point just defined. That is, the greatest fixed point of F under $\dot\sqsubseteq$ is $g\,\theta$.

This orderer θ is in fact a weak most general orderer of G. Any other orderer ρ satisfies $s_i\rho \dot\sqsupseteq t_i\rho$ for all i, so $f_i(g\rho) = g\rho$, and $F(g\rho) = g\rho$. Thus $g\rho$ is a fixed point of F. But $g\,\theta$ is the greatest such fixed point under $\dot\sqsubseteq$.

The fixed point is attained by F^k, for some *finite* k, since any unifier σ for G is an orderer for G, and any Term mgo θ satisfies $\theta\mu = \sigma$ for some substitution μ. That is, if σ is an idempotent mgu such that

$$[s_1,...,s_n]\sigma = [t_1,...,t_n]\sigma,$$

then σ gives a bound on the size of (number of symbols in) a Term mgo for G, since it is also an orderer of G.

Each application of F either increases the size of the orderer obtained, or leaves it undisturbed (in which case the fixed point has been reached). Since the size of the Term mgo is bounded, we must arrive at a greatest fixed point in a finite number of applications, and the fixed point approach here can be used as an algorithm to compute a mgo. □

A Herbrand mgo is just a Term mgo with the additional property that it can be extended to a grounding substitution without violating any constraints. It is easy to determine whether a Term mgo is also a Herbrand mgo:

Theorem 2b
A Term most general orderer θ is also a Herbrand most general orderer of

$$G = s_1 \sqsupseteq t_1, \cdots, s_n \sqsupseteq t_n$$

if and only if the two terms $[s_1,...,s_n]$ and $[t_1,...,t_n]$ are unifiable.
Proof

If σ is unifier of $[s_1,...,s_n]$ and $[t_1,...,t_n]$, there is a substitution ρ so that, for each i, $s_i\sigma\rho$ and $t_i\sigma\rho$ are ground and equal. So $\sigma\rho$ is a ground orderer of G. Thus there must be a Term mgo θ of G for which some γ exists such that $\sigma\rho = \theta\gamma$. But then θ must be a Herbrand mgo, since $\theta\gamma$ is also a ground orderer for G.

Conversely, assume G has a Term mgo θ that is also a Herbrand mgo. Then there is a substitution σ such that $G\theta\sigma$ is made ground and satisfied. But then $\theta\sigma$ is a unifier of $[s_1,...,s_n]$ and $[t_1,...,t_n]$. □

A conjunction of subsumption constraints

$$G = s_1 \sqsupseteq t_1, \cdots, s_n \sqsupseteq t_n$$

is a particular kind of *partial order program* [22]. These programs have applications of independent interest. For example, we will show later how collections of $\sqsupseteq$ constraints are sufficient for specifying polymorphic types for predicate arguments in logic programs. In other words, they are sufficient for inferring basic type and mode declarations for logic programs. Also, as indicated above they are solvable by fixed point iteration, or relaxation as it is often called.

3. Directed Logic Programs

In this section we define directed logic programs and show how directed programs relate to ordinary logic programs. We assume the reader is familiar with basic concepts in the semantics of logic programs. A more complete reference for ordinary logic programs is [15].

3.1. Definition of Directed Logic Programs

Every clause in a logic program

$$p(t_1, ..., t_n) \leftarrow G$$

can be written

$$p(V_1, ..., V_n) \leftarrow (V_1 = t_1), \ ..., \ (V_n = t_n), \ G.$$

where $V_1, ..., V_n$ are newly introduced variables, or equivalently

$$p(V_1, ..., V_n) \leftarrow (V_1 \sqsupseteq t_1, \ t_1 \sqsupseteq V_1), \ ..., \ (V_n \sqsupseteq t_n, \ t_n \sqsupseteq V_n), \ G.$$

A clause is *directed* if instead of $(V_i = t_i)$ it contains only $(V_i \sqsupseteq t_i)$ or $(t_i \sqsupseteq V_i)$ for one or more values of i, $1 \leq i \leq n$. Thus a directed clause uses a subsumption constraint instead of unification for at least one of its arguments. We identify the ith argument of p in the clause as an *input argument, output argument,* or both if its corresponding constraint is as follows:

input argument	$(V_i \sqsupseteq t_i)$
output argument	$(t_i \sqsupseteq V_i)$
input and output argument	$(V_i \sqsupseteq t_i, \ t_i \sqsupseteq V_i)$.

A *directed logic program* is a logic program containing at least one directed clause. Directed logic programs as defined here do *not* include a clausal definition of the predicate $\sqsupseteq$, since (as we will show below) $\sqsupseteq$ is not specifiable with Horn clauses.

The *undirected version* P_u of a directed logic program P is the program obtained by replacing all occurrences of $\sqsupseteq$ in P by = (unification), and including the clause

$$X = X \leftarrow .$$

3.2. Semantics of Directed Logic Programs

Corresponding to the two notions of satisfiability (Term-satisfiability and Herbrand-satisfiability) introduced earlier, we now define two notions of entailment and two notions of provability. In many situations Herbrand-satisfiability is more appropriate, but Term-satisfiability is desirable for type inference, so we continue our investigation with both.

Let P be a directed logic program, and G be a conjunction of atoms, so $\leftarrow G$ is a goal. We say $P \models_{Term} G$ if $P \cup \{ \leftarrow G \}$ is not Term-satisfiable, and $P \models_{Herbrand} G$ if $P \cup \{ \leftarrow G \}$ is not Herbrand-satisfiable.

Here $\models_{Herbrand}$ is the usual Herbrand model-theoretic notion of entailment, and it differs slightly from $\models_{Term}$. As an example, consider the program P containing the single predicate:

$$variable(X) \leftarrow X \sqsupseteq a, X \sqsupseteq b$$

This predicate has no Herbrand models! Under the definitions above we have

$$P \models_{Term} variable(Z)$$
$$P \not\models_{Herbrand} variable(Z).$$

Since the choice of Term- or Herbrand-satisfiability is usually implicit from context, below we omit the subscript on $\models$.
A *correct answer* for P and $\leftarrow G$ is a substitution θ such that $P \models G\theta$.

Now let us develop a proof theory by extending SLD resolution slightly to deal with $\sqsupseteq$. An extension is needed since $\sqsupseteq$ is not specifiable with Horn clauses: The goal $\leftarrow X \sqsupseteq a$ should succeed with the empty answer substitution, but if $\sqsupseteq$ is specified with Horn clauses the goal's ordinary SLD resolvent must then be the empty clause. However, the goal $\leftarrow X \sqsupseteq a,\ X = b$ should then also succeed. So $\sqsupseteq$ is not specifiable with Horn clauses.

Derivations can be defined as for Prolog II [7] and CLP [10]. We say the sequence

$$\leftarrow G_0,\ \leftarrow G_1,\ \ldots,\ \leftarrow G_n$$

is an *SLD*($\sqsupseteq$)*-derivation* from $\leftarrow G_0$ and P if, for each i between 1 and n, one of the following holds:

(1) G_i is the resolvent of G_{i-1} and some clause in P. Thus if

$$G_{i-1} = A_1, \ldots, A_j, \ldots, A_k$$

and we select a clause $A \leftarrow B_1, \ldots, B_q$ from P where an mgu of A and A_j is θ_i, then

$$G_i = (A_1, \ldots, A_{j-1}, B_1, \ldots, B_q, A_{j+1}, \ldots, A_k)\,\theta_i.$$

(2) G_i is the result of enforcing a subsumption constraint of G_{i-1}. If

$$G_{i-1} = A_1, \ldots, A_j, \ldots, A_k$$

and $A_j = (s \sqsupseteq t)$ is selected, then

$$G_i = (A_1, \ldots, A_j, \ldots, A_k)\,\theta_{[s,t]},$$

where as usual $\theta_{[s,t]}$ is an impartial mgo of s and t. If we permit an arbitrary orderer to be used instead of an mgo, we call the derivation an *unrestricted SLD*($\sqsupseteq$)*-derivation.*

An *SLD(⊒)-refutation* from $\leftarrow G_0$ and P is a finite SLD(⊒)-derivation $\leftarrow G_0, \ldots, \leftarrow G_n$ in which G_n is

$$s_1 \sqsupseteq t_1, \ \ldots, \ s_m \sqsupseteq t_m$$

and every constraint in this conjunction is satisfied. (Note: if m is 0, we get the definition [15] of ordinary SLD-refutation.) More precisely, we call the refutation a *SLD(⊒)-refutation using Term-satisfiability* if G_n is Term-satisfied, and *SLD(⊒)-refutation using Herbrand-satisfiability* if G_n is Herbrand-satisfied.

The *answer substitution* of the refutation is the substitution

$$\theta \ = \ \theta_1\theta_2 \cdots \theta_n$$

composing all unifiers obtained in the derivation.

Finally, we write $P \vdash G\theta$ if there is an SLD(⊒)-refutation from $\leftarrow G$ and P with answer substitution θ. Equivalently, there is an SLD(⊒)-refutation from $\leftarrow G\theta$ and P with answer substitution $\varnothing$. Where needed, we write $P \vdash_{Term} G\theta$ or $P \vdash_{Herbrand} G\theta$, respectively, to make precise that the final conjunction of constraints is Term-satisfiable or Herbrand-satisfiable.

Theorem 3
Let P_u be the undirected version of a directed logic program P. If $P_u \models G\theta$, then $P \models G\theta$. Also, if $P_u \vdash G\theta$, then $P \vdash G\theta$.
Proof

The theorem follows since equality implies subsumption. That is, $(s = t)$ is equivalent to $(s \sqsupseteq t) \wedge (t \sqsupseteq s)$, so $(s = t)$ logically implies $(s \sqsupseteq t)$, and $\leftarrow (s \sqsupseteq t)$ logically implies $\leftarrow (s = t)$. Consequently any directed clause

$$P \leftarrow \ (s \sqsupseteq t), \ (G) \ \equiv \ \left[P \vee \leftarrow(s \sqsupseteq t) \vee \leftarrow(G) \right]$$

logically implies

$$\left[P \vee \leftarrow(s = t) \vee \leftarrow(G) \right] \ \equiv \ P \leftarrow \ (s = t), \ (G)$$

Thus, every directed clause logically implies its undirected version, and so every directed program logically implies its undirected version. Consequently if $P_u \models G\theta$ then $P \models G\theta$.

Also, if $P_u \vdash G\theta$, then $P \vdash G\theta$. If $G_0 = G\theta$, any SLD-refutation $\leftarrow G_0, \ldots, \leftarrow G_n$ from $\leftarrow G\theta$ and P_u immediately yields an SLD(⊒)-refutation from $\leftarrow G\theta$ and P in which all subsumption constraints are satisfied in advance, i.e., no mgo $\theta_{[s,t]}$ must be used in the derivation.

Furthermore, if $G_0 = G$, any SLD-refutation $\leftarrow G_0, \ldots, \leftarrow G_n$ from $\leftarrow G$ and P_u yields an unrestricted SLD(⊒)-refutation in which every substitution applied to enforce a subsumption constraint is an orderer, but not necessarily a most general orderer. Consequently every subsumption constraint is satisfied by θ. □

We say a goal $(\leftarrow G)$ *respects the direction* of a directed program P if for any answer

substitution θ such that $P \vdash G\theta$, then also $P_u \vdash G\theta$. Intuitively, a directed logic program behaves exactly like its undirected version when the arguments are used in the proper directions.

3.3. Example

Consider the following directed program:

$$merge(V1,V2,V3) \leftarrow \; V1 \sqsupseteq [\,], \quad V2 \sqsupseteq B, \quad V3 \sqsubseteq B.$$
$$merge(V1,V2,V3) \leftarrow \; V1 \sqsupseteq A, \quad V2 \sqsupseteq [\,], \quad V3 \sqsubseteq A.$$
$$merge(V1,V2,V3) \leftarrow \; V1 \sqsupseteq [X \mid A], \quad V2 \sqsupseteq B, \quad V3 \sqsubseteq [X \mid C], \quad merge(B,A,C).$$

In this program the input variables $V1$ and $V2$ constrain the terms that the original program would unify them against, and $V3$ is constrained by the result of the merge. An SLD($\sqsupseteq$)-derivation from the goal

$$\leftarrow merge([a],[b,c],Z).$$

and this program results in the conjunction of the following constraints:

$V1_1$	$=$	$[a]$	$V1_2$	$=$	B_1	$V1_3$	$=$	B_2
$V2_1$	$=$	$[b,c]$	$V2_2$	$=$	A_1	$V2_3$	$=$	A_2
$V3_1$	$=$	Z	$V3_2$	$=$	C_1	$V3_3$	$=$	C_2
$V1_1$	$\sqsupseteq$	$[X_1 \mid A_1]$	$V1_2$	$\sqsupseteq$	$[X_2 \mid A_2]$	$V1_3$	$\sqsupseteq$	$[\,]$
$V2_1$	$\sqsupseteq$	B_1	$V2_2$	$\sqsupseteq$	B_2	$V2_3$	$\sqsupseteq$	B_3
$V3_1$	$\sqsubseteq$	$[X_1 \mid C_1]$	$V3_2$	$\sqsubseteq$	$[X_2 \mid C_2]$	$V3_3$	$\sqsubseteq$	B_3

We have included equality constraints here for clarity. Subscripts on variables indicate the step in which they were introduced. This conjunction of constraints has as its unique solution the usual substitution obtained by a logic program with equality (ordinary unification) constraints, i.e., with the undirected version of the *merge*. It does so because the goal respects the direction of *merge*. So there is an SLD($\sqsupseteq$)-refutation from the goal.

On the other hand, the goal

$$\leftarrow merge([X],[X],[a,b])$$

has an SLD($\sqsupseteq$)-refutation using Term-satisfiability from the directed program! This refutation does not bind X, since $(X \sqsupseteq a) \wedge (X \sqsupseteq b)$ is already Term-satisfied. However, the goal has no SLD($\sqsupseteq$)-refutation using Herbrand-satisfiability. Also, this goal will fail with the undirected version of the program, i.e., there is no SLD($\sqsupseteq$)-refutation from this goal and the undirected version of the program. The goal does not respect the direction of the program, since the input arguments to *merge* are not ground and have the variable X in common.

3.4. The Problem of Respect

The discussion above raises the question of how we can solve the following problem:

Given a directed logic program P and a goal $\leftarrow G$, does $\leftarrow G$ respect the direction of P?

This problem seems undecidable in general. A complete answer to the question of how to solve the problem is an interesting open issue. Note that Gregory [9] defines directional programs and discusses basic issues involving this problem and results like those of Theorem 3.

For the moment we content ourselves with a partial answer that covers many useful situations. A simple sufficient condition for $\leftarrow G$ to respect the direction of P is that in the final result

$$\leftarrow (s_1 \supseteq t_1),\ \ldots,\ (s_m \supseteq t_m)$$

of every SLD($\supseteq$)-refutation from ($\leftarrow G$) and P, every s_i is ground. This follows since $\supseteq$ is equivalent to = when its first argument is ground.

4. Implementation of Term Subsumption Constraints and the Importance of Types

This section describes how to apply the formalism developed above in real implementations of $\sqsupseteq$ and directed logic programming systems.

4.1. Implementing Term Subsumption in Prolog

A sketch of how $\sqsupseteq$ can be implemented in Prolog is as follows:†

```
Y⊑X  :-  X ⊒Y.
X⊒Y  :-   subsume_now(X,Y), subsume_henceforth(X,Y).
subsume_now(X,Y) :- term_variables(Y,L), term_copy(X,L,T), unify(T,Y).
subsume_henceforth(X,Y) :- ground(X), !.
subsume_henceforth(X,Y) :- ...
```

This implementation of (X $\sqsupseteq$ Y) first forces X to be more general than Y at the point of invocation, by unifying Y with T, a copy of X that retains all variables appearing in Y. (As we remarked at the end of section 2.3, this unification is equivalent to applying a most general orderer to X and Y.) The implementation then must constrain future bindings to X and Y.

In the implementation of (X $\sqsupseteq$ Y), when X is a ground term the future generality constraint `subsume_henceforth(X,Y)` is trivially satisfied. However, if X is not ground, the constraint (X $\sqsupseteq$ Y) must be enforced for any future bindings to X. Enforcement of this kind generally requires *suspension* of execution. Concurrent Prolog and PARLOG suspend on unbound input variables, MU-Prolog suspends negated goals [18], and both Prolog II [8] and CLP [11] defer such constraints until they become bound. In the same way, (X $\sqsupseteq$ Y) can suspend when X contains a variable.

The implementation discussion here is not really intended to cover committed choice languages such as PARLOG. In these languages one often wants execution of the current *process* to suspend until X becomes nonvariable. In fact, suspension of the current process is ultimately *necessary* to guarantee determinism of execution in processing guards and commitment. However it seems that the 'flat' versions of these committed choice languages could be covered by a similar kind of suspension.

Prolog II's `freeze` primitive [5,7] provides a basic mechanism for suspension. The goal `freeze(X,G)` normally suspends the goal G until X is nonvariable. Since it provides basically what we need, let us continue the implementation sketch with it.

It is easiest to develop a complete implementation for the constraint (X $\sqsupseteq$ Y) by considering three different restrictions that can be known to hold on the argument X:

(1) *Eventually Ground Terms.*
If X is known always to be either variable or ground, we can write:

† This and subsequent Prolog implementations all use `unify` (unification with the occur check) and not Prolog's '=' (unification without the occur check). Some applications of $\sqsupseteq$, such as type inference, explicitly require the occur check.

```
X ⊒ Y   :-   freeze(X, unify(X,Y)).
```

This restriction arises frequently in ordinary functional computations [2,4,19,20,21].

(2) *Eventually Terms with Independent Variables.*
If X is known always to be either variable or a term in which a variable can never appear more than once, and X is also known to never have have any variables in common with Y, then whenever X is of the form $f(s_1, \ldots, s_n)$, Y must be $f(t_1, \ldots, t_n)$, where $s_i \sqsupseteq t_i$ for $1 \le i \le n$. In other words, in this case the constraint X ⊒ Y can be decomposed into the independent constraints $s_i \sqsupseteq t_i$, $1 \le i \le n$. This may be implemented naively as follows:

```
X ⊒ Y   :-   freeze(X, subsume(X,Y)).

subsume(X,Y)   :-   functor(X,F,N),
                    functor(Y,F,N),
                    subsumeArgs(N,X,Y).

subsumeArgs(0,_,_).
subsumeArgs(I,X,Y)  :- 0 < I, arg(I,X,Xi), arg(I,Y,Yi), Xi ⊒ Yi,
                       I1 is I-1, subsumeArgs(I1,X,Y).
```

This restriction arises commonly in stream computations, where X will be bound to a list-like structure $[s_1|s_2]$ in which all variables in s_1 and s_2 are independent of one another.†

(3) *Arbitrary Terms.*
Otherwise, if X is unrestricted, implementation of subsume_henceforth is more challenging. The challenge comes from goals like

```
?-   f(A,B) ⊒ f(C,D),   A = B.
```

which should result in the binding C = D. This can be achieved only with extensions to the Prolog unifier that permit monitoring and enforcement of ⊒ constraints. Implementation requires something more powerful than freeze, like the general delay primitive [5] which activates its delayed goal whenever the variable upon which the delay is done is bound to any value, variable or nonvariable. A sample implementation in Prolog using delay is given in the Appendix. This implementation enforces only Term-satisfiability of subsumption constraints. Extending it to enforce Herbrand-satisfiability would not be difficult.

This discussion shows the importance of *typing* on the implementation of ⊒. When an input argument is eventually a ground term, or eventually a term with independent variables, efficient versions of ⊒ can be used.

---

† The implementation shown is very like that of PARLOG [9] for the one-way unification primitive:

```
c1 <= c2   <-   EQCONST(c1,c2):;
v1 <= v2   <-   EQVAR(v1,v2):;
v <= t   <- LOCKVAR(v) : BIND(v,t);
str1 <= str2   <-  (DATA(str1) & STRLIST(str1,list1)),
                   (DATA(str2) & STRLIST(str2,list2)) :
         list1 <= list2;
[h1|t1] <= [h2|t2]   <-   h1 <= h2, t1 <= t2.
```

4.2. Argument Typing and Streams

With the previous section in mind, we can define a *stream* to be either a term of type (1), or a term of type (2) with a fixed function symbol (e.g., [_|_]) whose arguments are, in turn, streams. This simple definition generalizes list-oriented stream structures. The implementations sketched above will give efficient directed stream-processing Prolog programs for streams of this kind. This definition is like Tribble et al's *channel* [26], a stream-like object providing partial ordering among data items. Now let us clarify how we can exploit argument typing information.

Fortunately, term subsumption can serve as a typing mechanism! In fact, subsumption constraints are often used precisely as typing constraints. Mycroft and O'Keefe [17] present a polymorphic typing system for Prolog based upon subsumption constraints. Chou [6] has applied the system to show how type inference for a Prolog program reduces to solution of a set of subsumption constraints on types, and how these constraints can be solved by relaxation. We briefly review the system and its application here.

Given a program P, a *typing* of P consists of a set of *type attachments* assigning a non-extended type to each term and subterm in P, and *type premises* assigning a type term to each predicate p, functor f, and variable X of P. These premises are written respectively as

$$\begin{aligned} p &: <\tau_1, \ldots, \tau_n> \\ f &: <\tau_1, \ldots, \tau_n> \rightarrow \tau_{n+1} \\ X &: \tau. \end{aligned}$$

A typing of P is a *well-typing* if it satisfies the following conditions. Let $p(t_1, \ldots, t_n)$ be a predicate and arguments appearing somewhere in a clause of P. If the typing has type premise $p : <\tau_1, \ldots, \tau_n>$ for p and assigns types $\sigma_1, \ldots, \sigma_n$ to the arguments $t_1, \ldots, t_n$, then:

(1) When $p(t_1, \ldots, t_n)$ is the *head* of the clause, the typing must satisfy

$$<\tau_1, \ldots, \tau_n> \; = \; <\sigma_1, \ldots, \sigma_n>$$

In a well-typing, every clause agrees with the typing of its predicate.

(2) When $p(t_1, \ldots, t_n)$ is in the *body* of the clause, the typing must satisfy

$$<\tau_1, \ldots, \tau_n> \; \sqsupseteq \; <\sigma_1, \ldots, \sigma_n>.$$

In a well-typing, every use of a predicate obeys the typing of the predicate.

For example, consider the following Prolog program:

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Assume we begin with types for [] and [_|_], and the corresponding type premises:

`[] :`	$\rightarrow list(T)$
`[_|_] :`	$<T, list(T)> \rightarrow list(T)$
`append :`	$<A, B, C>$
`L :`	D
`X :`	E
`L1 :`	F
`L2 :`	G
`L3 :`	H

The unknown types $A, B, ..., H$ are then constrained as follows:

$$
\begin{array}{lcl}
<A, B, C> & = & <list(T), D, D> \\
<A, B, C> & = & <I, G, J> \\
<T, list(T)> \rightarrow list(T) & \sqsupseteq & <E, F> \rightarrow I \\
<T, list(T)> \rightarrow list(T) & \sqsupseteq & <E, H> \rightarrow J \\
<A, B, C> & \sqsupseteq & <F, G, H>
\end{array}
$$

For this conjunction of constraints, the bindings { *A / list(T), B / list(T), C / list(T)* } give a well-typing. The type of `append` is correspondingly inferred to be

$$< list(T), list(T), list(T) >.$$

This approach can be used to define streams, and to deal with the *problem of respect* defined earlier for stream processing programs. We simply define *stream* to be another type like *list(T)*. Mycroft and O'Keefe [17] suggest type descriptions for functors of the form

type list(T) => [], [*T* | *list(T)*].

To continue the discussion, let us make the further definition

type stream => *list(ground)*

where *ground* is the type of all ground terms. With this definition we then *constrain* the types of all stream arguments to obey corresponding type subsumption relationships. A goal and program that meet the type subsumption constraints will respect the direction of the program.

For example, consider the following version of the merge program investigated in the previous section:

```
merge(V11,V21,V31) :- V11 ⊒ [],    V21 ⊒ B,  V31 ⊑ B.
merge(V12,V22,V32) :- V12 ⊒ A,     V22 ⊒ [], V32 ⊑ A.
merge(V13,V23,V33) :- V13 ⊒ [X|L], V23 ⊒ Y,  V33 ⊑ [X|Z], merge(L,Y,Z).
```

This program yields the following type constraints:

$$\begin{array}{lcl}
<V1, V2, V3> & = & <V11, V21, V31> \\
V11 & \sqsupseteq & list(T) \\
V21 & \sqsupseteq & B \\
V31 & \sqsubseteq & B \\
<V1, V2, V3> & = & <V12, V22, V32> \\
V12 & \sqsupseteq & A \\
V22 & \sqsupseteq & list(T) \\
V32 & \sqsubseteq & A \\
<V1, V2, V3> & = & <V13, V23, V33> \\
<T, list(T)> \rightarrow list(T) & \sqsupseteq & <X, L> \rightarrow R \\
<T, list(T)> \rightarrow list(T) & \sqsupseteq & <X, Z> \rightarrow S \\
V13 & \sqsupseteq & R \\
V23 & \sqsupseteq & list(T) \\
V33 & \sqsubseteq & S \\
<V1, V2, V3> & \sqsupseteq & <L, Y, Z>
\end{array}$$

This system of inequalities reduces to:

$$\begin{array}{lcl@{\qquad}lcl}
L & = & list(X) & A & \sqsupseteq & list(X) \\
R & = & list(X) & B & \sqsupseteq & list(X) \\
S & = & list(X) & T & \sqsupseteq & X \\
V31 & = & list(X) & V1 & \sqsupseteq & list(T) \\
V32 & = & list(X) & V1 & \sqsupseteq & A \\
V33 & = & list(X) & V2 & \sqsupseteq & list(T) \\
V3 & = & list(X) & V2 & \sqsupseteq & B \\
Z & = & list(X) & V2 & \sqsupseteq & Y
\end{array}$$

The type of `merge` is $< V1, V2, V3 >$ where from the above we can extract $V1 \sqsupseteq list(T)$, $V2 \sqsupseteq list(T)$, $V3 \sqsubseteq list(T)$. Now, if we add the information that the first two arguments are known to be instances of streams,

$$V1 \sqsubseteq list(ground)$$
$$V2 \sqsubseteq list(ground),$$

then the type of `merge` is inferred to be $< list(ground), list(ground), list(ground) >$, i.e., $< stream, stream, stream >$.

The Mycroft-O'Keefe system permits static declaration of type constraints, and is appropriate for use with an optimizing compiler. In some situations it is desirable to permit *dynamic* specification of type constraints, as with the Prolog goal

```
X isa stream,
```

and allow these constraints to be included directly in programs. In the next section we investigate generalizations of the subsumption mechanism to handle partial orders beyond the subsumption ordering we have investigated thus far.

5. Extensions

The foregoing addresses the use of the first-order term partial ordering within the context of logic programming. This discussion can now be extended to permit more general ordering constraints defined for more specific argument types.

For example, if arguments are known to be functional expressions (terms representing the application of a function to some arguments) a reduction ordering can be used. Similarly, if arguments are members of a predefined type hierarchy, a type inclusion ordering can be used.

Existing work has explored a variety of useful ordering constraints. Jaffar et al's Constraint Logic Programming incorporates linear equality and inequality constraints [10, 11], and Ait-Kaci and Nasr's system generalizes first-order terms and unification to a partial order (more precisely, a distributive lattice) of ψ-terms and a meet operator on these terms [1, 3]. Recently this system has been extended to include functional terms and suspension of evaluation [2, 4].

In this paper we will concentrate on the Log(F) system of Narain [19, 20, 21], since it has immediate applications in stream processing. Log(F) is an efficient integration of functional programming with Prolog. Log(F) programs appear as collections of rewrite rules:

```
append([],B) => B.
append([X|A],B) => [X|append(A,B)].
```

Log(F) rules are compiled into Prolog clauses in a straightforward way. For example, the two rules above are translated into something functionally equivalent to the following Prolog code:

```
reduce( [], [] ).
reduce( [H|T], [H|T] ).
reduce(append(V1,V2),B) :- reduce(V1, []), reduce(V2,B).
reduce(append(V1,B),V3) :- reduce(V1, [X|A]), reduce([X|append(A,B)],V3).
```

The `reduce` rules here can operate nondeterministically, and can be used directly with Prolog. With the rules above, the goal

```
?- reduce( append([1,2,3],[4,5,6]), X ).
```

yields the result

```
X = [1|append([2,3],[4,5,6])].
```

That is, in one `reduce` step, the head of the resulting appended list is computed. The tail, `append([2,3],[4,5,6])`, can be further `reduced` if necessary. Thus Log(F) also provides lazy evaluation.

The formal foundation of Log(F) accommodates stream processing:

(a) Log(F) makes the basic requirement that terms to be `reduced` are ground terms. Furthermore, left hand sides of all rules are required to be linear, i.e., to not include duplicate variables. This avoids issues encountered by parallel Prolog systems in providing consistency of bindings to variables used by processes on opposite ends of streams.

(b) Determinate code is easily detected using only syntactic tests. Rules that backtrack can be localized, avoiding distributed backtracking where it is not absolutely necessary.

These features of Log(F) make it a nicely-limited Prolog sublanguage in which to write high-powered programs for stream processing and other performance-critical tasks.

Log(F) reductions can be treated like subsumption constraints. The insights described earlier for directed logic programs can also be applied to Log(F) code. The only difference is that the partial ordering of subsumption among first-order terms is replaced by the partial ordering (or more generally a preordering) defined by `reduce`, which is a rewrite or functional evaluation ordering. In other words, we can write directed logic programs using `reduce` rather than $\sqsupseteq$.

A naive Prolog merge predicate, for example, whose first and second arguments are input streams and whose third argument is an output stream, can use the `reduce` predicate on its arguments:

```
merge(V1,V2,V3) :- reduce(V1,[]),     reduce(V2,B),   reduce(B,V3).
merge(V1,V2,V3) :- reduce(V1,A),      reduce(V2,[]),  reduce(A,V3).
merge(V1,V2,V3) :- reduce(V1,[X|A]),  reduce(V2,B),   reduce([X|C],V3),
                   merge(B,A,C).
```

Thus we can type arguments to predicates as of 'Log(F) type', and being either input or output. With these arguments, the `reduce` ordering can be used, giving at the same time functional evaluation and stream consumption.

In other words, we are led to the definition of a *general directed logic program* as a logic program whose clauses can be put in the form

$$p(V_1, ..., V_n) \leftarrow \pi_1(V_1, t_1), ..., \pi_n(V_n, t_n), G.$$

where the variables V_i appear only as shown, and each π_i is a binary predicate defining a partial order (or at least a preorder) constraint such as $\sqsubseteq$, $\sqsupseteq$, =, `reduce`, etc. It appears that general directed logic programs appear quite frequently in practice, and clarifying their properties will be important.

6. Conclusion

We have presented a theory of directedness in logic programming systems. The theory replaces unification with partial ordering constraints on predicate arguments, making the arguments 'directed'. When enough is known about an argument's type, efficient implementation of these constraints is possible. The resulting system retains theoretical elegance and may attain higher performance than is possible with a system required to support (undirected) unification.

Since directedness manifests itself in both stream and functional computations, the theory gives insight in a number of important contexts. A stream is naturally defined here to be any structure, or term, about which certain instantiation patterns are guaranteed. The *Tangram* environment makes heavy use of streams and functional operations on streams, so the theory is of definite practical significance.

Many interesting further avenues for work are available at this point:

(1) The precise relationship between $\sqsubseteq$ and PARLOG's one-way unification operator <= must be clarified. Applications of our notion of directedness in committed-choice languages should also be investigated.

(2) The extension of the theory for subsumption on infinite (rational) terms [7] is of immediate concern for implementation of $\sqsubseteq$ in Prolog systems.

(3) Clearly the directedness and variable instantiation patterns of predicate arguments give only a limited notion of typing. Extension of this approach for other types and ordering constraints is of pressing interest as well. For example, there are many different types of streams: partially ordered streams, backtrackable streams, lookahead-access streams, multiaccess streams, time-series streams, streams of streams, unreliable streams, etc.

There is a lot to be done here.

Acknowledgement

Paul Eggert provided extensive comments and suggestions that have reshaped this paper in a number of ways. His standards are a source of inspiration. Mike Gorlick, Shen-Tzay Huang, Richard Huntsinger, Carl Kesselman, Brian Livezey, Sanjai Narain, and Tom Page also provided a number of important improvements on the manuscript.

Appendix: An Implementation of General First-Order Term Subsumption

Below is a listing of a simple implementation of the term inequality constraint ⊒ using the delayed execution primitives `freeze/2`, `frozen/2`, and `delay/2` [5]. This implementation runs successfully on SICS Prolog, distributed by the Swedish Institute of Computer Science. It is not particularly efficient. It is presented as a sketch of how ⊒ can be implemented, and as a starting point for further investigation by the reader.

```
%-----------------------------------------------------------------------------
%    Term Subsumption Constraints
%
%    X >== Y  enforces the constraint that X must be more general than Y.
%    The constraint is enforced by unifying Y with less general terms.
%
%    The goal   ?- X>==Y, Y>==X.   results in X and Y being unified.
%
%    Current implementation uses delay primitives provided by SICS Prolog.
%-----------------------------------------------------------------------------

:- op(700,xfx, >== ).
:- op(700,xfx, <== ).

X <== Y :- Y >== X.

X >== Y :- var(X), !, mergeFrozenGoals(X,Y).
X >== Y :- atomic(X), !, X=Y.
X >== Y :- subsume(X,Y,X,Y).


%-----------------------------------------------------------------------------
% subsume(+Xsubterm,+Ysubterm,+Xterm,+Yterm)
%    sets up constraints    Xsubterm >== Ysubterm   that are implicit in
%    the original constraint   Xterm >== Yterm.
%    These constraints need to be set up only where Xsubterm is variable.
%
%    User unification of variables in Xterm should cause unification of the
%    corresponding variables in Yterm.  This is implemented with delays.
%    All variables in Xterm have variable tests set up.
%-----------------------------------------------------------------------------

subsume(S,T,X,Y) :- var(S), !, mergeFrozenGoals(S,T), varTest(S,T,X,Y).
subsume(S,T,_,_) :- atomic(S), !, S = T.
subsume(S,T,X,Y) :- functor(S,F,N), functor(T,F,N), subsume(S,T,X,Y,0,N).

subsume(_,_,_,_,N,N) :- !.
subsume(S,T,X,Y,I0,N) :-
     I is I0+1,
     arg(I,S,Si),
     arg(I,T,Ti),
     subsume(Si,Ti,X,Y),
     subsume(S,T,X,Y,I,N).
```

```
%-----------------------------------------------------------------------------
% varTest(+Xsubterm,+Ysubterm,+Xterm,+Yterm)
%    This predicate is normally delayed when Xsubterm is a variable,
%    and is then invoked when Xsubterm is (nontrivially) unified against.
%    We need four arguments here since usually a constraint  Xterm >== Yterm
%    is not fully decomposable into constraints on subterms.  Ex:  Although
%    f(S,T) >== f(U,V)  implies both  S >== U  and  T >== V,  the binding
%    S = T  requires us enforce the global constraint by unifying U and V.
%
% 1. If Xsubterm is variable, Yterm must be inspected for occurrences of
%    Xsubterm.  If any are found, so Xterm and Yterm share the variable
%    Xsubterm, then Xsubterm must be unified with Ysubterm (unless they
%    are already identical) in order to satisfy  :- (Xterm >== Yterm).
%    [The mgo theta such that  (Xterm)theta >== (Yterm)theta  will have some
%    idempotent rho such that  ((Xterm)theta)rho = (Yterm)theta,  where rho
%    does not affect shared variables of Xterm and Yterm.  Thus the most
%    general orderer must make all necessary bindings to shared variables.]
%    Ex: Xsubterm = S, Xterm = f(S,T), Yterm = f(a,S)  =>  S and a are unified.
%
% 2. If Xsubterm is still variable, then all of Xterm must be searched for
%    other matching occurrences of Xsubterm that may have appeared recently,
%    and corresponding occurrences of Ysubterm must be unified with each other.
%    Ex: Xsubterm = S, Xterm = f(S,S), Yterm = f(U,V)  =>  U and V are unified.
%    Afterwards we must again delay, waiting for further bindings to Xsubterm.
%
% 3. If Xsubterm has become a structure, Xsubterm >== Ysubterm is enforced.
%    Then all variables in Xsubterm are constrained to subsume the
%    corresponding part of Ysubterm.  In the process of doing this,
%    all variables in Xsubterm have varTest done for them as well.
%-----------------------------------------------------------------------------

varTest(S,T,X,Y) :- unifyIfNecessary(S,T,X,Y), checkSubsume(S,T,X,Y).

unifyIfNecessary(S,T,_,Y) :- var(S), S \== T, occursIn(S,Y), !, unify(S,T).
unifyIfNecessary(_,_,_,_).

checkSubsume(S,T,X,Y) :- var(S), !, checkMatch(X,Y,S,_), delay(S,varTest(S,T,X,Y)).
checkSubsume(S,_,_,_) :- atomic(S), !.
checkSubsume(S,T,X,Y) :- subsume(S,T,X,Y).

checkMatch(X,Y,V,W) :- var(X), !, matchCorrespondingSubterm(X,Y,V,W).
checkMatch(X,_,_,_) :- atomic(X), !.
checkMatch(X,Y,V,W) :- functor(X,F,N), functor(Y,F,N), checkMatch(X,Y,V,W,0,N).

checkMatch(_,_,_,_,N,N) :- !.
checkMatch(X,Y,V,W,I0,N) :-
     I is I0+1,
     arg(I,X,Xi),
     arg(I,Y,Yi),
     checkMatch(Xi,Yi,V,W),
     checkMatch(X,Y,V,W,I,N).

matchCorrespondingSubterm(X,Y,V,W) :- V == X, !, unify(W,Y).
matchCorrespondingSubterm(_,_,_,_).
```

```
%-----------------------------------------------------------------------------
%  Frozen goal management
%
%  mergeFrozenGoals(X,Y) determines if  X >== Y  can be simplified (X is var):
%  1.  If  X >== Y  has already been specified, the constraint is ignored.
%  2.  If  X >== Y, Y is variable, and Y >== X is inferrable, then X=Y.
%  3.  Otherwise we freeze  X >== Y.  Freezing is sufficient here, since
%      this predicate need not handle variable matching constraints as above.
%-----------------------------------------------------------------------------

mergeFrozenGoals(X,Y) :- frozen(X,FGs), memberGoal(freeze(X,X>==Y),FGs), !.
mergeFrozenGoals(X,Y) :- var(Y), equatedVars(X,Y), !.
mergeFrozenGoals(X,Y) :- freeze(X,X>==Y).

memberGoal(G,(A,_)) :- memberGoal(G,A).
memberGoal(G,(_,B)) :- !, memberGoal(G,B).
memberGoal(G,A) :- G == A.

equatedVars(X,Y) :- X == Y, !.
equatedVars(X,Y) :- varsBetween(Y,X,[],L,1), unifyList(L,X).

varsBetween(U,V,L,L, 1) :- U == V, !.  % variable V reached from U successfully
varsBetween(U,_,L,L, 0) :- memberchkVar(U,L), !.  % cycle detected, failure
varsBetween(U,V,L,L1,1) :- frozen(U,FGs), varsB(FGs,V,[U|L],L1,1), !.
varsBetween(_,_,L,L, 0).

varsB((G,H),V,L,L2,F) :- !, varsB(G,V,L,L1,A), varsB(H,V,L1,L2,B), F is A \/ B.
varsB(freeze(X,X>==Y),V,L,L1,F) :- var(Y), Y \== X, !, varsBetween(Y,V,L,L1,F).
varsB(_,_,L,L,0).

unifyList([],_) :- !.
unifyList([Y|L],X) :- X == Y, !,  unifyList(L,X).
unifyList([Y|L],X) :- unify(X,Y), unifyList(L,X).

memberchkVar(X,[Y|_]) :- X == Y, !.
memberchkVar(X,[_|L]) :- memberchkVar(X,L).

delay(X,G) :- 'SYSCALL'('$geler'(X,G)).
% like freeze/2, but calls G when X is bound to anything, even a variable.


%-----------------------------------------------------------------------------
%  Utilities
%  In the interest of performance some users may prefer the definition:
%  unify(X,X).
%-----------------------------------------------------------------------------

unify(X,Y) :- var(X), var(Y), !, X=Y.
unify(X,Y) :- var(X), !, occursCheck(X,Y), X=Y.
unify(X,Y) :- var(Y), !, occursCheck(Y,X), Y=X.
unify(X,Y) :- functor(X,F,N), functor(Y,F,N), unifyArgs(N,X,Y).

unifyArgs(0,_,_) :- !.
unifyArgs(N,X,Y) :-
```

```
     arg(N,X,A),
     arg(N,Y,B),
     unify(A,B),
     N1 is N-1,
     unifyArgs(N1,X,Y).

occursCheck(V,T) :- \+ occursIn(V,T).

occursIn(V,T) :- var(T), !, V == T.
occursIn(V,T) :- atomic(T), !, fail.
occursIn(V,T) :- functor(T,_,N), occursInArgs(V,T,0,N).

occursInArgs(V,T,I0,N) :-
     I0 < N,
     I is I0+1,
     arg(I,T,A),
     occursInArgument(V,T,I,N,A).

occursInArgument(V,_,_,_,A) :- occursIn(V,A), !.
occursInArgument(V,T,I,N,_) :- occursInArgs(V,T,I,N).
```

References

1. Ait-Kaci, H., ''A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures,'' Ph.D. dissertation, University of Pennsylvania, Dept. of Computer and Information Science, Philadelphia, PA, 1984.
2. Ait-Kaci, H. and R. Nasr, ''Residuation: A Paradigm for Integrating Logic and Functional Programming,'' Technical Report AI-359-86, MCC, October 1986.
3. Ait-Kaci, H. and R. Nasr, ''LOGIN: A Logic Programming Language with Built-in Inheritance,'' *Journal of Logic Programming*, vol. 3, no. 3, pp. 185-215, October 1986.
4. Ait-Kaci, H., P. Lincoln, and R. Nasr, ''Le Fun: Logic, equations, and Functions,'' *Proc. Symp. on Logic Programming*, pp. 17-23, IEEE Computer Society #799, September 1987.
5. Carlsson, M., ''Freeze, Indexing, and Other Implementation Issues in the WAM,'' *Proc. 4th Intnl. Conf. on Logic Programming*, pp. 40-58, MIT Press, Melbourne, Australia, May 1987.
6. Chou, C.-T., ''Relaxation Processes: Theory, Case Studies and Applications,'' Report CSD-860057 (M.S. Thesis), UCLA Computer Science Dept., Los Angeles, CA, February 1986.
7. Colmerauer, A., H. Kanoui, and M. van Caneghem, ''Prolog, theoretical principles and current trends,'' *Technology and Science of Informatics*, vol. 2, no. 4, pp. 255-292, 1983.
8. Colmerauer, A., ''Equations and Inequations on Finite and Infinite Trees,'' *Proc. Intnl. Conf. on Fifth Generation Computer Systems (FGCS'84)*, pp. 85-99, North-Holland, Tokyo, November 1984.
9. Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1987.
10. Jaffar, J. and J-L. Lassez, ''Constraint Logic Programming,'' *Proc. 12th ACM Symposium on Principles of Programming Languages*, pp. 111-119, Munich, West Germany, January 1987.
11. Jaffar, J. and S. Michaylov, ''Methodology and Implementation of a CLP System,'' *Proc. 4th Intnl. Conf. on Logic Programming*, pp. 196-218, MIT Press, Melbourne, Australia, May 1987.
12. Lassez, J-L., V.L. Nguyen, and E.A. Sonenberg, ''Fixed Point Theorems and Semantics: A Folk Tale,'' *Information Processing Letters*, vol. 14, no. 3, pp. 112-116, 16 May 1982.
13. Lassez, J-L., M.J. Maher, and K.G. Marriott, ''Unification Revisited,'' in *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker, pp. 587-625, Morgan Kaufmann Publishers, Los Altos, CA, 1988.
14. Li, P.-Y.P. and A.J. Martin, ''The Sync Model: A Parallel Execution Method for Logic Programming,'' *Proc. Symp. on Logic Programming*, pp. 223-234, Salt Lake City, 1986.
15. Lloyd, J., *Foundations of Logic Programming, 2nd Edition*, Springer-Verlag, New York, 1987.

16. Muntz, R.R. and D.S Parker, ''Tangram: Project Overview,'' Technical Report CSD-880032, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, April 1988.

17. Mycroft, A. and R.A. O'Keefe, ''A Polymorphic Type System for Prolog,'' *Artificial Intelligence*, vol. 23, 1984.

18. Naish, L., ''Negation and Control in Prolog,'' LNCS #238, Springer-Verlag, New York, 1986.

19. Narain, S., ''A Technique for Doing Lazy Evaluation in Logic,'' *J. Logic Programming*, vol. 3, no. 3, pp. 259-276, October 1986.

20. Narain, S., ''LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation,'' Technical Report CSD-870027, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

21. Narain, S., ''LOG(F): An optimal combination of logic programming, rewrite rules and lazy evaluation,'' Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, forthcoming, 1988.

22. Parker, D.S., ''Partial Order Programming,'' Technical Report CSD-870067, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

23. Parker, D.S., T.W. Page, and R.R. Muntz, ''Improving Clause Access in Prolog,'' Technical Report CSD-880024, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

24. Plotkin, G.D., ''A Note on Inductive Generalization,'' in *Machine Intelligence 5*, ed. B. Meltzer, D. Michie, pp. 153-163, J. Wiley/Halstead Press, New York, 1970.

25. Reynolds, J.C., ''Transformational Systems and the Algebraic Structure of Atomic Formulas,'' in *Machine Intelligence 5*, ed. B. Meltzer, D. Michie, pp. 135-151, J. Wiley/Halstead Press, New York, 1970.

26. Tribble, E.D., M.S. Miller, K. Kahn, D.G. Bobrow, and C. Abbott, ''Channels: A Generalization of Streams,'' in *Concurrent Prolog: Collected Papers, vol. 1*, ed. E. Shapiro, pp. 446-463, MIT Press, 1987.

27. Wheeler, T.A., ''Unification and Predicate Locking,'' Technical Report CSD-880023 (M.S. Thesis), UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1986. Issued as a Technical Report, April 1988.