Abstract Interpretation of Logic Programs

Renganathan Sundararajan
e-mail: renga@cs.uoregon.edu

CIS-TR-91-05
December 91

Abstract

This paper is intended to be an introduction to the field of abstract interpretation of logic programs. Abstract interpretation can be thought of as providing a new (or abstract) semantics for the language under consideration and then showing how the meaning of a program in the new semantics is an abstraction of the meaning of the program in standard semantics. We discuss the issues involved in deriving an abstract interpretation scheme and present an efficient and domain-independent algorithm for computing fixed points in the context of abstract interpretation of logic programs. A brief introduction to denotational semantics is also provided in order to make the paper self-contained.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Contents

1 Overview **2**

2 Introduction to Denotational Semantics **2**

2.1 Functions as first class objects 3
2.2 Curried Functions 4
2.3 Lambda Notation 5
2.4 Denotational Definitions - A simple example 6
2.4.1 Recursive Functions 7
2.5 Domain Theory 9
2.6 Domain Constructors 12
2.7 Recursive Domains 14

3 Logic Programming **14**

3.1 Abstract Syntax 14
3.2 Semantic Domains 16
3.2.1 Review of SLD-Resolution 17
3.3 Valuation Functions 18
3.4 An Example 19

4 Abstract Interpretation **22**

4.1 Core Semantics 23
4.2 Conditions on *Asub*, *entry_sub*, and *success_sub* 25
4.3 Correctness 25
4.4 Ground Interpretation I_{ground} 26

5 Design and Implementation Issues **30**

5.1 Data Abstraction 30
5.2 Abstract Domain Operations 31
5.3 Fixed Point Computation 32
5.3.1 The Naive Method 32
5.3.2 Minimal Function Graph Strategy 34
5.3.3 An algorithm for MFG computation 35
5.4 Correctness and Termination 39

6 Extensions and Summary **40**

1 Overview

Abstract Interpretation is a formal approach to deriving properties of programs. Compile time analyses within the framework of abstract interpretation cover many areas such as source to source transformations [24], software validation, strictness analysis, reference counting, and flow analysis [1].

The field of abstract interpretation has grown considerably in the last decade. Although the theory was developed originally for analyzing programs written in procedural languages [7], most of the recent work in the field has concentrated on declarative languages.

This paper is intended to be an introduction to the field of abstract interpretation of logic programs. Abstract interpretation can be thought of as providing a new (or abstract) semantics for the language under consideration and then showing how the meaning of a program in the new semantics is an abstraction of the meaning of the program in standard semantics. As is the case with any abstraction, in defining an abstract semantics for a language one pays attention to aspects that are relevant to the purpose at hand and ignores others. Abstract semantics is usually defined in such a way that the meaning of any program is computable in a finite (usually short) amount of time.

In order to make this paper more or less self-contained and the material understandable, we provide a brief introduction to denotational semantics. As an illustration of abstract interpretation, we show how some of the properties of pure logic programs can be derived. We discuss the issues involved in deriving an abstract interpretation scheme and present an efficient and domain-independent algorithm for computing fixed points in the context of abstract interpretation of logic programs. We assume the reader is familiar with the fundamentals of logic programming including resolution. If not, [18] is a good introduction to the subject. In a companion paper [27], we present an abstract interpretation scheme for efficient and precise derivation of such properties as independence, groundness, sharing and freeness of logic variables.

2 Introduction to Denotational Semantics

Semantics of a programming language may be defined using operational, axiomatic or denotational approaches. In all approaches, the meaning of a program in a given language is defined in terms of the meanings of the language constructs. The meaning of a program is usually the composition of the meanings of its sub-parts.

An operational semantics for a language is defined by giving an implementation independent interpreter, *an abstract machine*, for the language. The meaning of language constructs is then defined in terms of the instructions and transformations of the state space associated with the abstract machine. The abstract machine instructions and the state space must be simple and precise, leaving little scope for ambiguity.

In the axiomatic approach, certain assertions, called *input* and *output assertions*, or *pre* and *post conditions*, are associated with each language construct. Input assertions are assumed to be true just before the execution of a language construct. The meaning of a construct is defined by giving (output) assertions that are true immediately after its execution. The language used for specifying input and output assertions is usually first order predicate calculus.

In the denotational approach to semantics, meanings of language constructs are defined by *valuation* or *semantic* functions. Valuation functions map syntactic constructs to semantic value spaces such as numbers, truth values, and functions. The meaning of a construct is defined as a composition of the meanings *denoted* by its proper sub-parts.

We need a notation to define the valuation functions and value spaces. Lambda calculus is the language most often used for this purpose. A brief introduction to higher order functions (or functions as first class objects), curried functions and lambda notation is provided in this section as a prelude to the discussion on denotational semantics. A more in-depth discussion of the same material can be found in [26].

2.1 Functions as first class objects

In order to understand lambda notation and appreciate the rationale behind it, one must first deal with functions as first class objects. We explain below what we mean by first class objects.

Many popular programming languages can be studied in two parts. There is the applicative or descriptive part which is used to build expressions, and there is the command or imperative part which is used to specify state transformations and control flow. One can see a dichotomy in the treatment of expressions. Arithmetic and logic expressions may be assigned to variables, passed as parameters to functions and procedures, returned as results of function invocations, selected by conditional expressions, and stored in arrays. All this can be done without *naming* the expressions. In order to use a function, one must not only define a function but also name it. Functions may be passed as parameters but not returned as results. In the words of Stoy, they are *under-privileged* objects.

In the expression $(x + y) \times (x - y)$, the symbols x and y are variables, i.e., they denote some value from the space of say, integers. But, the symbols $+$ and $-$ are *not* variables denoting some function from the space of functions that map an integer pair to an integer. Each denotes a *specific* function. Even when there is overloading of function/operator symbols, the overloading may usually be resolved from context. Likewise, in the expression $f(x, y)$, f is regarded as a constant standing for a specific function and x and y are variables. As mentioned before, x and y may have functions as their values but f may not return a function as the result. In a language in which functions are first-class objects, we may write *fxy* instead of *f(x,y)* to emphasize that f may be a variable ranging over functions, just as x and y may be. If we use juxtaposition to mean function application, we run into a difficulty. Does *fxy* mean *f(x,y)* or *f(x(y))* or *(f(x))(y)*? We solve this problem by using *curried functions* and an implied rule of associativity which are discussed next.

2.2 Curried Functions

To any function of two or more arguments, there exists an equivalent set of functions that take one argument at a time. Such a function is known as a curried function, named after H.B.Curry who made use of this technique introduced by Schonfinkel [26]. Let $f(x,y)$ be a function of two arguments. Then its curried version is a function f' such that $(f'(x))(y) = f(x,y)$, i.e., f' is a function of one argument x that returns an unnamed function, which when applied to y, produces a result that is equivalent to $f(x,y)$. Notice that we have started treating functions as first class objects. First, f' returns a function. Second, we did not *name* that function in order to use it, i.e., apply it to y. We don't need to, just as we don't name sub-expressions in an expression. Since we will be using only curried functions, we no longer need to use parentheses around the arguments. With the understanding that function application associates to the left, we will write $f'xy$ to mean $((f'x)y)$.

Functions are specified in two parts: the first part gives the type or functionality and second part describes the actual mapping. The mapping may be described by an equation, a set (graph of a function), or a table. We will use equational definitions most of the time but occasionally we may list part of the mapping (i.e., the argument, result pairs) to understand the function intuitively. Let us look at a familiar example. We define *Add* to be a function that adds two natural numbers and returns the result, a natural number. To indicate the functionality of *Add*, we write

$$Add: \quad N \times N \rightarrow N$$

Some times we may assign a meaningful name to the above functionality, and use it wherever the functionality may be used. For example, we may write

$$Add: \quad \text{PairToN} = N \times N \rightarrow N$$

to mean that PairToN is the set of all functions that map a pair of natural numbers to a natural number and that *Add* is one such function. In subsequent sections, we impose some constraints on such sets and call them *domains*. Let *Add'* be the curried version of *Add*. Its functionality is

$$Add': \quad N \rightarrow (N \rightarrow N)$$

By convention, the function builder $\rightarrow$ associates to the right and we denote the type of *Add'* as:

$$Add': \quad N \rightarrow N \rightarrow N$$

What is the value of $Add'\,x$, where $x \in N$? It is a function which takes an argument $y \in N$ and returns $x+y$ as the result. To make things clear, let us look at the graphs of *Add* and *Add'*. *Add* has the following graph.

$$\begin{array}{lllll}
\{ & ((0,0),0), & ((0,1),1), & ((0,2),2), & \ldots \\
 & ((1,0),1), & ((1,1),2), & ((1,2),3), & \ldots \\
 & & \vdots & & \\
\} & & & &
\end{array}$$

The graph of *Add* is an infinite set. Each element of this set is an ordered pair of argument and result. For example, *Add* maps (0,0) to 0, (0,1) to 1 and so on. The graph of *Add'* is

$$\begin{array}{ll}
\{ & (0,\{(0,0),(1,1),(2,2),\ldots\}), \\
 & (1,\{(0,1),(1,2),(2,3),\ldots\}), \\
 & \vdots \\
\} &
\end{array}$$

The graph of *Add'* is also an infinite set of ordered pairs of argument and result. In the case of *Add*, an argument is a pair of natural numbers. For *Add'*, it is just a natural number. The result of *Add* is a natural number obtained by adding the first and second element of the ordered pair. *Add'*, on the other hand, maps a natural number, say *i*, to a *function* which in turn maps a natural number *j* to *j+i*.

2.3 Lambda Notation

In the previous section, we used curried functions and a left associativity rule for function application to avoid any ambiguity. But we did not specify a syntax for defining functions. We use lambda notation introduced by Church [26] for this purpose. An *expression* in λ-notation is defined as follows.

```
expr  ::=  variable            |
           expr expr           |
           λ variable . expr   |
           (expr)
```

The first two rules are simple to understand. A variable is a valid expression. Juxtaposition of two expressions, *fx*, denotes the application of function *f* to *x*. An expression *fxy* is traditionally taken to mean *(fx)y*, i.e., function application associates to the left. The fourth rule allows one to override the default left associativity of function application. The third rule introduces something new. It defines an operation called λ-abstraction or simply abstraction. If M is an expression denoting values from some set D_2, and x is a variable ranging over the set D_1, then the expression $\lambda x.M$ denotes the function $f : D_1 \rightarrow D_2$ such that for any $d \in D_1$, *fd* can be obtained by evaluating M using the value d for x. In programming language terms, x is the formal parameter of the function defined by the expression $\lambda x.M$, and M is the body of the function. We will not rigorously define the phrase *evaluating M using the value d for x*, since it would take us into the realm of substitution rules, and free and bound variables. Instead, we rely on intuition and direct the curious reader to [26].

2.4 Denotational Definitions - A simple example

A denotational definition of a language consists of three parts: abstract syntax, semantic value spaces, and valuation functions. Abstract syntax defines the structure of sentences of a language at the level of *words* or *tokens*. The syntax rules defining the form of individual words (or tokens) of a language are not considered. For example, an identifier may be defined as a string of alphabets or numerals, starting with an alphabet. Such definitions do not deal with the sentence structure but with the structure of individual words. Examples of semantic value spaces are numbers, truth values, ordered pairs, and functions over these spaces. A valuation function maps an abstract syntactic construct to a semantic value space.

The syntactic and semantic value spaces of many practical programming languages are defined recursively. In order to give meanings to the recursive definitions of programs and data types, domain theory is used. Domain theory, developed by Dana Scott, is about sets called *domains* and functions that are general enough to model computation, including recursion and self-application. We can think of a domain as a set together with a structure or a partial ordering imposed on its elements.

We will also be interested in the operations defined for a domain. For a simple domain like N, the operations may be plus, times etc. Before defining domains formally, we will motivate their use with examples. Consider the language of binary numerals defined by the following BNF rules, where <zero> and <one> are the tokens corresponding to the numerals *0* and *1*.

B:	Binary Numeral	::=	BD
B:	Binary Numeral	::=	D
D:	Binary Digit	::=	<zero>
D:	Binary Digit	::=	<one>

A binary numeral B is defined to be a binary digit D or a B followed by a D. A binary digit D is either a <zero> or a <one>. The meaning of a binary numeral is simply the natural number it denotes. The semantic value space is N together with the operations times and plus. The elements of N are *zero, one, two, ...* We use boldface letter **A** (by convention) to denote the semantic function associated with a syntactic category A. A semantic function is defined by a set of equations, one for each option of the BNF rule defining the syntactic category. The category B has two BNF rules and hence two semantic equations. It is customary to enclose a syntactic object in $[\![\,]\!]$ when it appears in semantic equations.

The valuation functions have the following functionality

B: Binary Numeral → N
D: Binary Digit → N

and are defined as

B⟦BD⟧	=	(**B**⟦B⟧ *times two*) *plus* **D**⟦D⟧
B⟦D⟧	=	**D**⟦D⟧
D⟦<zero>⟧	=	*zero*
D⟦<one>⟧	=	*one*

Note that the meaning of a syntactic object is defined either directly (i.e., mapped to an object in the semantic domain) or as the result of some operation(s) applied to the meanings of its sub-parts. In the case of functions, we would like to capture the *mapping* of a function as its denotation. In the next two sections, we develop this intuitive idea further, leading towards the formalization of the denotation of a function as the least fixed point of the associated functional. The terms "fixed point" and "associated functional" will be explained shortly.

2.4.1 Recursive Functions

Consider the following specification of a function q:

$$q : N \rightarrow N_\perp$$
$$q = \lambda x.\, x \; \textit{equals zero} \rightarrow \textit{one} \mid q(x \; \textit{plus one})$$

The symbol $N_\perp$ stands for the domain of Natural numbers augmented by the least element $\perp$ (pronounced *bottom*). $\perp$ can be thought of as denoting the "undefined value" or equivalently, non-termination. The domain construction operation of adding $\perp$ to a domain is known as *lifting*. This and other domain construction operations are discussed later, in section 2.6.

According to this specification, q maps zero to one. For all arguments $i > 0$, all that the specification says is $q(i) = q(i \; \textit{plus one})$. The specification is silent about the value of $q(i)$ for $i > 0$. One may choose some $k \in N$ or $\perp$ to be that value and still satisfy the specification. In the operational treatment of recursion, evaluation of $q(i)$ for any $i > 0$ does not terminate. Since $\perp$ models non-termination, choosing $\perp$ to be the value of the function for all $i > 0$ corresponds to the operational treatment of recursion. Moreover, this choice yields the function with the least arbitrary information as compared to all other choices. This notion of *least information* will be formalized later as the *least fixed point.* Let us consider another example, the recursive specification of *factorial.*

$$\textit{factorial} : N \rightarrow N_\perp$$
$$\textit{factorial}(n) = \lambda n.\, n \; \textit{equals zero} \rightarrow \textit{one} \mid n \; \textit{times} \; (\textit{factorial}(n \; \textit{minus one}))$$

Unlike the previous specification, there is only one function that corresponds to this specification. The graph of this function is the infinite set

$$\{(zero, one), (one, one), (two, two), (three, six), ...(i, i!), ...\}.$$

Although it is an infinite set, we can produce any particular element of this set in a finite number of steps (i.e., it is recursively enumerable). To find the value of *factorial*(i), for any $i \in N$, we don't have to construct this infinite object. All we have to do is to unfold the recursive specification $i + 1$ times. For example, if we unfold the specification zero times, then the corresponding graph is {}, the empty set. If we unfold it once, the graph is {(*zero*, *one*)}. In general, if we unfold the specification n+1 times, the graph looks like

$$\{(zero, one), (one, one), \ldots (n, n!)\}.$$

It is notable that each unfolding yields a graph that wholly contains the graphs of all previous unfoldings. In other words, each successive unfolding, which we now call approximation, is

- consistent with the previous approximations and
- contains more information.

The graph produced at i^{th} unfolding defines a finite sub-function *factorial*$_i$. It is clear that,

$$\forall i \geq 0,\ graph(factorial_i) \subseteq graph(factorial_{i+1}) \text{ and}$$
$$\forall i \geq 0,\ graph(factorial_i) \subseteq graph(factorial),$$

i.e., each *factorial*$_i$ is consistent with the ultimate solution to the specification. It is easy to prove that

$$graph(factorial) = \bigcup_{i=0}^{\infty} graph(factorial_i) \qquad (1)$$

It means that *factorial* can be understood solely in terms of the finite sub-functions. It is the union of the meanings of the finite sub-functions. Each sub-function *factorial*$_i$ can be defined and understood non-recursively in terms of the sub-functions *factorial*$_j$, $0 \leq j < i$. Define *factorial*$_i$, for $i \geq 0$ as:

$$factorial_i : N \rightarrow N_\perp$$
$$factorial_0 = \lambda n.\perp$$
$$factorial_i = \lambda n.\ n\ equals\ zero \rightarrow one \mid n\ times\ factorial_{i-1}(n\ minus\ one),\ \forall i > 0.$$

Each *factorial*$_i$ is non-recursive and we now have an infinite family of finite sub-functions. The next step is to extract the common form from their definitions to yield a functional F and then show that *factorial* is a fixed point of F. A function (such as F, to be defined soon) that takes a function as an argument and produces another function as its result is called a *functional.*

What is common to all the finite sub-functions of *factorial*? It is the free identifier *factorial*$_{i-1}$ in the definition of *factorial*$_i$ (i.e., in programming language terms, neither a parameter to the function nor locally defined). Let us define a new function F by converting the free identifier to a parameter f.

$$F : (N \rightarrow N_\perp) \rightarrow (N \rightarrow N_\perp)$$
$$F = \lambda f.\, \lambda n.\, n\ \mathit{equal\ zero} \rightarrow \mathit{one} \mid n\ \mathit{times}\,(f(n\ \mathit{minus\ one})).$$

Compare the definitions of F and $\mathit{factorial}_i$. The only difference is that F has an additional parameter f which is used in the body of F exactly where $\mathit{factorial}_{i-1}$ was used in the body of $\mathit{factorial}_i$. It is clear that

$$\mathit{factorial}_i = F(\mathit{factorial}_{i-1}),\ \forall i > 0 \tag{2}$$

We have defined all the finite sub-functions $\mathit{factorial}_i$ in terms of F and we know that *factorial* is just the infinite union of all $\mathit{factorial}_i$. So it should be possible to define *factorial* in terms of F. From equations 1 and 2, it can be shown that

$$\mathit{graph}(\mathit{factorial}) = \bigcup_{i=0}^{\infty} \mathit{graph}(F^i(\lambda n.\perp)).$$

Furthermore, if we apply F to *factorial*, we will see that *factorial* is a fixed point of F, i.e.,

$$F(\mathit{factorial}) = \mathit{factorial}$$

If $g(x) = x$ for some function g, then x is called the fixed point of g. F is what we have been calling the "associated functional" of *factorial*. Any recursive function f can be defined as $f = F(f)$ and the meaning of f is defined to be the least fixed point of F. We have shown that the meaning of *factorial* is a fixed point of the associated functional F. The notion of *leastness* is formalized in the following section, after introducing some basic notions from domain theory.

A function may not have a fixed point or may have more than one fixed point. In the case of function q which we discussed at the beginning of this section, the *least fixed point* maps non-zero arguments to $\perp$. All other fixed points contain more information (i.e., map non-zero arguments to some arbitrary $n \in N$) not warranted by the specification. Thus it seems reasonable to take the least fixed point of the associated functional of a function to be the denotation or meaning of that function.

2.5 Domain Theory

The domains we are interested in have partial elements approximating total (or limit) elements. In the previous section, we arrived at the meaning of *factorial* as the infinite union (or the limit) of all finite approximations (or partial elements) $\mathit{factorial}_i$. Hence we require that such limits exist in our domains. A domain must also have a least element so that we can compute any approximation in a finite number of steps by starting with the least element and computing each successive approximation from its predecessor. Functions over domains must be *continuous.* Continuous functions allow us to describe finite approximations of infinite objects, such as the function *factorial.* We will explain this after introducing some terminology from basic set theory.

Definition
A partially ordered set, *poset*, is a pair $(D, \sqsubseteq)$ where D is a set and $\sqsubseteq$ is a binary relation (on D) that is reflexive, anti-symmetric and transitive. □

$\sqsubseteq$ is pronounced as *less defined than* or *approximates.* Some times we use $\sqsubseteq_D$ to indicate that we are talking about the partial ordering defined on D, but when there is no room for confusion, the subscript D will be omitted.

Example 1 $(\mathcal{P}(Q), \subseteq)$ is a poset, where $Q = \{1,2,3,4\}$ and $\mathcal{P}(Q)$ is its powerset.

We can talk about elements being less defined than sets and vice versa. Let $(D, \sqsubseteq)$ be a poset, $X \subseteq D$ and $u, l \in D$.

Definition
u is an *upper bound* of X $(X \sqsubseteq u)$ iff $\forall x \in X, \quad x \sqsubseteq u$.
l is a *lower bound* of X $(l \sqsubseteq X)$ iff $\forall x \in X, \quad l \sqsubseteq x$. □

Using the above example, $\{1\}$ is a lower bound of $\{\{1,2\} \{1,2,3\}\}$ and $\{1,2,3,4\}$ is an upper bound of $\{\{1,2\} \{1,2,3\}\}$.

If A is the set of all upper bounds of X, the least element of A (that element $a_0 \in A$ such that $a_0 \sqsubseteq a_i$ for every element $a_i \in A$) is called the least upper bound (lub) of X and denoted as $\sqcup X$. For example, $\sqcup X = \{1, 2, 3\}$ where $X = \{\{1,2\} \{1,2,3\}\}$. When we take the lub of a set containing two elements, say x and y, we may write $x \sqcup y$ instead of $\sqcup\{x, y\}$.

Definition
A nonempty $X \subseteq D$ is a *chain* of D iff X is totally ordered, i.e., for any $a, b \in X, a \sqsubseteq b$ or $b \sqsubseteq a$. □

The set $X = \{\{1\}, \{1, 2\}, \{1, 2, 4\}, \{1, 2, 3, 4\}\} \subseteq \mathcal{P}(Q)$ is a chain, since any two elements of X are related (or comparable).

Definition
A poset $(D, \sqsubseteq)$ is a *complete partial order*, cpo, iff the lub of a chain X is in D for every chain $X \subseteq D$. □

It is easy to see that $\mathcal{P}(Q)$ is a cpo. Here is an example of a poset which is not a cpo. Let R be the set of all *finite* subsets of N. $(R, \subseteq)$ is a poset. The least upper bound of the chain $\{\{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}, \{1,2,3,4,5\} \ldots\}$ is N itself and since N is not a member of R, R is not a cpo.

Definition

A cpo D is a *pointed cpo* iff there is an element $d \in D$ such that $d \sqsubseteq x \;\; \forall x \in D$. d is the least element of D. $\square$

Since the empty set ϕ is less defined than all other elements of $\mathcal{P}(Q)$, it is the least element and $\mathcal{P}(Q)$ is pointed cpo.

Definition

For cpos A and B, a function $f : A \rightarrow B$ is *continuous* iff for any chain $X \subseteq A$, $f(\sqcup X) = \sqcup\{f(x)|x \in X\}$. $\square$

In other words, the image (under f) of the least upper bound of a chain X contains the same information (or is the same) as the least upper bound of the images (under f) of the elements of X. Continuous functions preserve the limits of chains.

Definition

A function $f : A \rightarrow B$ is *monotonic* iff $\forall x, y \in A \;\; x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$. $\square$

Continuity implies monotonicity and when the underlying set is finite, monotonicity and continuity are equivalent. Continuous functions allow us to compute finite approximations of infinite objects. We may not be able to store and process *F(f)* for some infinite object *f*. But if *f* is the least upper bound of some chain $\{f_i\}$ and each f_i is finite, then we can store each f_i successively and apply F to it. The value *F(f)* can be constructed piece by piece as $F(f_0) \sqcup F(f_1) \sqcup F(f_2) \sqcup F(f_3) \sqcup \ldots$.

In the factorial example, the finite sub-functions $factorial_i$ constitute a chain, since $graph(factorial_i) \subseteq graph(factorial_{i+1})$. We cannot store and process the infinite mapping corresponding to the meaning of *factorial*. However, we can compute *factorial(i)* for any $i \in N$ by storing and computing the successive finite approximations $factorial_k$, $0 \leq k \leq i$.

We are now ready to define the axioms of Domain theory.

- A domain is a pointed cpo.
- Functions over domains are continuous.

The second axiom does not mean that all functions over domains are continuous. It just means that only continuous functions over domains are considered (as part of domain theory).

We now prove a theorem [26] that if D is a domain and $F : D \rightarrow D$ is a continuous function, then there exists a $d \in D$ such that

$F(d) = d$	d is a fixed point of F and
$\forall e \in D$, if $e = F(e)$ then $d \sqsubseteq e$	d is the least of all fixed points

The least fixed point d of F, *fix F*, is defined as

$$fix\ F = \sqcup\{F^i(\perp)|i \geq 0\}$$

where F^i is $F \circ F \circ \cdots F$, i times. *fix* is a fixed point operator. The proof of this fixed point theorem is given below for the sake of completeness.

Proof: First show that *fix F* is a fixed point of *F*.

$$\begin{array}{rcll}
F(\mathit{fix}\ F) & = & F(\sqcup\{F^i(\bot) \mid i \geq 0\}) & \text{by definition of } \mathit{fix}\ F \\
& = & \sqcup\{F(F^i(\bot)) \mid i \geq 0\} & \text{by continuity of } F \\
& = & \sqcup\{F^i(\bot) \mid i \geq 1\} & \\
& = & \sqcup\{F^i(\bot) \mid i \geq 0\} & \text{since } F^0(\bot) = \bot \text{ and } \bot \sqsubseteq F(\bot) \\
& = & \mathit{fix}\ F &
\end{array}$$

Next show that *fix F* is the least fixed point of *F*. Let e be a fixed point of *F*.

$$\begin{array}{lcll}
\bot & \sqsubseteq & e & \text{bottom is the least element} \\
F^i(\bot),\ \forall i \geq 0 & \sqsubseteq & F^i(e),\ \forall i \geq 0 & \text{by monotonicity} \\
F^i(\bot),\ \forall i \geq 0 & \sqsubseteq & e & \text{since } F(e) = e \\
\sqcup\{F^i(\bot) \mid i \geq 0\} & \sqsubseteq & e & \text{by definition of lub} \\
\mathit{fix}\ F & \sqsubseteq & e & \text{by definition of } \mathit{fix}\ F
\end{array}$$

Thus the denotation of a recursive specification $f = F(f)$ is taken to be the least fixed point *fix F* of *F*. For any recursive specification *f*, we can ***construct*** the corresponding functional *F* by following the method used in the *factorial* example and apply the fixed point operator *fix* to obtain its least fixed point.

Domain theory guarantees the existence of unique functions corresponding to our recursive specifications and these functions are the least, from an information theoretic point of view. The fact that they are the least also corresponds to the operational treatment of recursion. (See the discussion about function q at the beginning of this section).

2.6 Domain Constructors

Primitive Domains

A primitive domain such as N is a set whose elements are atomic and which is used in constructing other domains. Strictly speaking, N is not a domain according our definition since there is no least element (as defined by $\sqsubseteq$) in N. However it is trivial to add a least element to N as we will see. Another example of a primitive domain is the domain T of truth values. An element in a primitive domain has no more or no less information than any other element. For example, *zero* has the same amount of information as *one*. Primitive domains have a discrete partial ordering in the sense that each element in the domain is not related to any other element, i.e., $x \sqsubseteq y$ iff $x = y$.

Any set D with a discrete partial ordering is a cpo and any function $f : D \rightarrow E$ where E is some domain, is continuous. This is easy to prove. Since chains in D are singleton sets, the lubs of these chains exist in D and D is a cpo. The continuity of f can be established similarly.

Lifted Domains

For a poset $(A, \sqsubseteq_A)$, its *lifting* $A_\perp$ is the set $A \cup \{\perp\}$, partially ordered by the relation $\sqsubseteq$ such that for $x, y \in A_\perp$, $x \sqsubseteq y$ iff either $x = \perp$ or $x \sqsubseteq_A y$. If A is a cpo, then $A_\perp$ is a cpo and $f : A_\perp \rightarrow B_\perp$ is continuous when $f : A \rightarrow B$ is continuous. As mentioned before, the cpo N can be made into a pointed cpo by lifting. All the operations on the original domain carry over to the lifted domain. For proofs of this and subsequent claims about domain constructors preserving domain properties, the reader is referred to [23].

Product Domains

For posets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, their product $A \times B$ is the set $\{(a,b) \mid a \in A \wedge b \in B\}$ partially ordered by the relation $\sqsubseteq$ such that $(a, b) \sqsubseteq (a', b')$ iff $a \sqsubseteq_A a'$ and $b \sqsubseteq_B b'$. If A and B are (pointed) cpos, then $A \times B$ is a (pointed) cpo. The product domain has an assembly operation (called injection) to build an element of the product domain from two elements of the constituent domains and its inverse, a disassembly operation (called projection) and these are continuous. The product domain construction can be generalized to products of n domains, where $n > 2$. For a domain D, D^n denotes the product domain $D \times D \ldots n$ times. For a $d \in D_1 \times \cdots D_n$ and $1 \leq i \leq n$, $d{\downarrow}i$ refers to the i^{th} component of d, i.e., ${\downarrow}i$ is the projection operation.

Function Domains

For posets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, we define a function space $A \rightarrow B$ (read A *to* B) to be the set of all continuous functions with domain A and co-domain B. These functions are partially ordered by the relation $\sqsubseteq$ such that for $f, g : A \rightarrow B$,

$$f \sqsubseteq g \quad \text{iff} \quad \forall a \in A \;\; f(a) \sqsubseteq_B g(a).$$

In other words, we compare the functions pointwise. If A and B are cpos, then $A \rightarrow B$ is a cpo. The operations function abstraction and application are continuous.

As an example, the set of functions $\{factorial_0, factorial_1, factorial_2, \ldots, factorial\}$ (which is a subset of the set of all continuous functions $N \rightarrow N_\perp$) is a cpo and the partial ordering is $factorial_i \sqsubseteq factorial_{i+1}$ and $factorial_i \sqsubseteq factorial$ for all i. It is easy to see that $factorial_0$ is the least element of this domain since it is undefined everywhere. Each $factorial_{i+1}$ is as defined as $factorial_i$ at all points *zero* through *i-1* and is also defined at point i whereas $factorial_i$ returns $\perp$ for i.

Other Domains

We will introduce a few more domain constructions without explaining them in detail. If D is a domain, D^* denotes the domain of finite sequences of $d \in D$. If D is a set, then its power set $\mathcal{P}(D)$ is a domain with set inclusion (or set intersection) as the partial ordering. For other domain constructions such as disjoint-sum, see [23]. (Readers familiar with ML may note that ML's datatype declaration constructs a disjoint-sum domain out of component domains.)

2.7 Recursive Domains

We motivated the least fixed point construction by developing a series of non-recursive finite sub-functions that *approximated* the meaning of the recursive function. The meaning of the recursive function was then shown to be equivalent to the least upper bound of an infinite chain of finite sub-functions. The domains we used in the examples (N, $N \times N$, and $N_\perp$) are not recursive. If a language has recursive data types such as trees and lists, we will need recursive domains to define a denotational semantics for the language. We can give meanings to recursive domain specifications the same way we handled recursive function specifications.

The *inverse limit construction* method developed by Scott [23] finds a solution to a recursive domain specification. It is analogous to the way we construct the meaning of a recursive function specification. We build a series of approximating domains D_i. The approximating domains will be contained in the solution domain and each approximating domain D_i will be a subdomain of approximating domain D_{i+1}, i.e., the elements and the partial order structure of D_i will be preserved in D_{i+1}. The least element of this chain of approximating domains, D_0, is taken to be $\{\perp\}$. Clearly D_0 is a pointed cpo and each successive D_{i+1} constructed from D_i and the recursive specification will also be a pointed cpo. We will not go into the formal details of inverse limit construction method. For any recursive domain specification of the form $D = F(D)$ where F is an expression built with the domain constructors which preserve the domain properties, there is a domain D_∞ that is isomorphic to $F(D_\infty)$ and D_∞ is the least such cpo that satisfies the specification. The recursive domains (also called reflexive domains) are general enough to permit self-application without giving rise to paradoxes.

In conclusion, domains and continuous functions over domains are general enough to model computation and to give meanings to the syntactic entities called programs. For a detailed explanation of the ideas presented here, see [26, 19, 23, 28].

3 Logic Programming

In this section we provide a denotational semantics for a pure Horn Clause Logic programming language. We first define the abstract syntax of the language and then the semantic domains. After a brief review of SLD-resolution [18], we define the valuation functions. Finally, the meaning of a small program is derived using the valuation functions.

3.1 Abstract Syntax

We start with some definitions. In First Order Predicate Calculus, a literal is an atomic formula or the negation of an atomic formula. A disjunction of literals is a clause and a clause with at most one positive literal is a Horn Clause. A clause with exactly one positive literal is also known as a definite or program clause and a clause with no positive literal is a goal clause. In order to keep the presentation simple, the definition of a program (given below) includes a single goal clause, b_0. The positive literal of a clause is known as its head and the negative literals its body. In an atom $p(t_1, \ldots, t_n)$, p is the predicate symbol and $t_1 \ldots t_n$ are its arguments which are terms. A term t is either a variable or of the form $f(t_1, \ldots t_k)$ where f is a function symbol of arity k and $t_1, \ldots t_k$ are all terms. When k is zero, we omit the parentheses and f is called a constant. All the variables in a clause are universally quantified and hence we omit the quantifiers.

A logic program is a finite set of Horn Clauses. A set of clauses $\{c_1, c_2, \ldots, c_n\}$ stands for the conjunction $c_1 \wedge c_2 \cdots \wedge c_n$. Since $a \rightarrow b \equiv \neg a \vee b$, we will write $\{p \vee \neg q_1 \vee \neg q_2 \vee \ldots \vee \neg q_n\}$ as $p \leftarrow q_1, q_2, \ldots, q_n$. To summarize, the language has the following abstract syntax.

$$\begin{array}{llcll}
P: & \mathit{Prog} & ::= & \{h_1 \leftarrow b_1, \ldots, h_n \leftarrow b_n, \leftarrow b_0\} & (n \geq 1) \\
b: & \mathit{Body} & ::= & a_1, \ldots, a_p & (p \geq 0) \\
a, h: & \mathit{Atom} & ::= & p(t_1, \ldots, t_o) & (o \geq 0) \\
t: & \mathit{Term} & ::= & v \mid f(t_1, \ldots, t_q) & (q \geq 0)
\end{array}$$

The other syntactic categories are *Pred*, *Func*, and *Var* symbols whose members are p, f, and v respectively. These are assumed to be countably infinite and mutually disjoint.

Semantics

An operational semantics for a language is defined by giving an implementation independent interpreter for it. Fixed point semantics define the meaning of a program as the least fixed point of a transformation or functional associated with the program. Van Emden and Kowalski established in a seminal paper [11] the equivalence of these two semantics for Horn clause logic programs (provided the inference system used in defining the operational semantics is complete).

The denotational semantics presented in this section is a variation of the usual fixed point semantics [11] and is a simplified version of Jones and Sondergaard's exposition [15]. In this formulation, the meaning of a logic program is taken to be the input-output relation computed by the program. The derivation of the meaning of a program using the valuation functions defined later in this section may resemble SLD-resolution [18]. The purpose of the denotational definition is to emphasize the input-output relation computed by the program and to relate the abstract semantics (to be presented in the next section) to the standard semantics to be presented later in this section. and not to focus on the resolution process, which is operational.

We take a logic programming system to be a goal oriented system and hence the definition of a program includes a goal. Given a program, we are interested in the substitutions computed by the program for the variables in the goal(s) and the substitutions that prevailed at various points in the program during a top-down execution of the goal(s). The goal oriented or top-down approach is appropriate for many types of analyses, such as groundness analysis presented in this paper. For other applications like type inferencing, a bottom up approach may be more suitable.

3.2 Semantic Domains

A substitution σ : *Subst* = *Var* $\rightarrow$ *Term* is a mapping

$$\{v_i \mapsto t_i, \ldots, v_n \mapsto t_n\}$$

where the v_i are distinct elements of *Var*, t_i are elements of *Term* and v_i do not occur in any of the t_i. Discrete ordering is the partial ordering on the domain *Subst*. The definition of substitution can be extended naturally from *Var* $\rightarrow$ *Term* to *Term* $\rightarrow$ *Term*. (Hereafter we will use substitutions to mean either the original definition or its extension and the usage will be clear from the context.) The composition of substitutions $\sigma_j \circ \sigma_i$, is defined as

$$\lambda\, x.\, \sigma_j(\sigma_i\, x) \quad \text{where } x \in \mathit{Term}.$$

It can be shown that substitutions are idempotent, i.e., $\sigma \circ \sigma = \sigma$. The significance of idempotence is that applying a substitution one or more times to a term produces the same result. We may view a substitution as a bipartite graph because of this property and will do so in a later section. Define $var(\sigma) = dom(\sigma) \cup range(\sigma)$ where σ is a substitution and define $var(T)$ to be the set of variables in T, where T is a term. There is a natural pre-ordering of substitutions:[1]

$$\sigma_1 \leq \sigma_2 \text{ iff } \exists \sigma_3 \in \mathit{Subst} \text{ such that } \sigma_2 = \sigma_3 \circ \sigma_1.$$

Intuitively speaking, σ_1 is less specific than σ_2. A unifier σ of two terms t_1, t_2 is a substitution such that $\sigma t_1 = \sigma t_2$, i.e., the term obtained by applying the substitution σ_1 to t_1 is syntactically identical to the term obtained by applying σ_2 to t_2. The most general unifier of two terms is a unifier which is least with respect to $\leq$ (modulo variable renaming). Define a function $mgu : Atom \times Atom \rightarrow Subst \cup \{fail\}$ (where *fail* is a constant) that yields the most general unifier of two atoms if there is one and returns *fail* otherwise. Unification of first order terms is decidable and the unification algorithm [22] realizes the function *mgu*. We use the pre-ordering $\leq$ of substitutions solely to define the function mgu.

The denotation (or the meaning) D of an atom and a body will be defined to be a function that maps a set of substitutions to a set of substitutions. The partial ordering on $\mathcal{P}(Subst)$ is set inclusion and set union is the lub operation. To summarize, the semantic domains are:

θ :	*Subst*	Substitutions
Φ :	$\mathcal{P}(Subst)$	Sets of substitutions
d :	$D = \mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)$	Atom, Body Denotations

[1]A pre-ordering is a relation that is transitive and reflexive but not necessarily anti-symmetric.

We assume that the variables in a program have been renamed so that a variable does not occur in more than one clause. Hence we can refer unambiguously to the set of program variables *Pvar*. In order to avoid variable name clashes, we need a renaming function. The renaming function uses a renaming index which is a sequence of natural numbers denoting the depth of recursion. To keep the presentation simple, the renaming index is omitted from all the semantic functions in the rest of the paper.

3.2.1 Review of SLD-Resolution

Given a program P, a goal G = $\{a_i, \ldots, a_n\}$ and a current substitution θ, a subgoal a_m is selected using a *fair computation rule* [18] and removed from the set of subgoals $\{a_i, \ldots a_n\}$. A matching clause j is selected using a *search rule* [18], the variables of clause j are renamed, the head h_j of a clause j is unified with $\theta\, a_m$. If the unification succeeds, the body literals $b_{j1}, \ldots, b_{jk}$ are added to the set of goals to be solved and the new current substitution is $\theta' \circ \theta$, where θ' is the mgu of the head and the goal. The above process is repeated until it terminates because the set of goals is empty or there is a subgoal θa such that it does not unify with the head of any of the clauses. If the set of goals becomes empty, then the current substitution is one answer substitution corresponding to the initial substitution.

We assume that the goals in the body of a clause are solved sequentially in textual order and all the clauses whose heads match the current goal are tried in parallel. The parallel search rule just means that we are interested not in the *sequence* of solutions but in the set of solutions. The left-to-right computation rule is more troublesome and is dealt with in section 5.

To facilitate the understanding of valuation functions to be described, we wish to give different names to the current substitution θ depending on the state of computation. The initial substitution for the variables in the goal is denoted as ε.

- If a goal is about to be unified with the head of a clause, the current substitution (to be applied to the goal) is known as the *call substitution* of the goal.
- If a goal has just been successfully unified with the head of a clause (which has been renamed), the unifier is the *entry substitution* for the clause and also the call substitution for the first literal in the body of the clause.
- If the last literal in the body has just been solved, the current substitution is known as the *exit substitution* of the clause.
- Restricting the exit substitution of a clause i to the variables in the environment of goal a (which unified with the head of clause i) and then composing it with the call substitution of a gives us one *success substitution* of goal a with respect to clause i.

Given an entry substitution for a clause, its exit substitution is computed as follows. If it is a unit clause, the exit substitution is the same as the entry substitution. Otherwise, compute the success substitution of literal one, using the entry substitution of the clause as the call substitution. Using the success substitution of literal i as the call substitution of literal *i+1*, compute the success substitution of literal *i+1*. The success substitution of the last literal in the clause is the exit substitution of the clause. In the rest of the paper, we use $\theta, \theta', \ldots$ to denote substitutions and $\Phi, \Phi', \ldots$ to denote sets of substitutions.

3.3 Valuation Functions

We define three valuation functions, one for each of the syntactic categories *Prog, Atom,* and *Body*. The valuation function **P** maps a program P to a set of answer substitutions and the functions **B** and **A** map a body and an atom to their denotations D which are mappings from $\mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)$. We need an auxiliary 5-ary function, *den_atom*, which will be discussed later. The functionalities are as follows.

$$\begin{aligned}
\mathbf{P}: &\quad Prog \rightarrow \mathcal{P}(Subst)\\
\mathbf{A}: &\quad Prog \times Atom \rightarrow D\\
\mathbf{B}: &\quad Prog \times Body \rightarrow D\\
den\_atom: &\quad Prog \times Atom \times Atom \times Body \times \mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)
\end{aligned}$$

When the body of a clause is empty, the exit substitutions of that clause are the same as its entry substitutions. Thus the denotation of an empty body, [], is the identity function. This is described by the following equation. Remember that a syntactic object is enclosed in $[\![\,]\!]$ when it appears in a semantic equation.

$$\mathbf{B}[\![P]\!][\![[\,]]\!]\Phi \;=\; \Phi$$

The next semantic equation defines the meaning of a non-empty body recursively. The meaning of a non-empty body $a_i \ldots a_j$ is a function that maps a set of (entry) substitutions Φ to a set of (exit) substitutions Φ'' such that the denotation of atom a_i maps the set of (call) substitutions Φ to (success) substitutions Φ' and the denotation of the remainder of the body $a_{i+1}, \ldots a_j$ maps Φ' to Φ''. This is captured by the following equation.

$$\mathbf{B}[\![P]\!][\![a_i, \ldots, a_j]\!]\Phi \;=\; \mathbf{B}[\![P]\!][\![a_{i+1}, \ldots, a_j]\!]\Phi' \text{ where } \Phi' = \mathbf{A}[\![P]\!][\![a_i]\!]\Phi$$

When the body is a single literal, the above equation simplifies to

$$\mathbf{B}[\![P]\!][\![a]\!]\Phi \;=\; \mathbf{A}[\![P]\!][\![a]\!]\Phi$$

Valuation function **A** defines the denotation of an atom a in terms of an auxiliary function *den_atom*. *den_atom* defines the meaning of an atom a with respect to a particular clause j of the program. Let the call substitutions of a be Φ and let the denotation of an atom a with respect to a particular clause j map a and Φ to Φ'_j. Then the denotation of a maps Φ to the union of Φ'_j over all clauses j of the program. In the following equation, h_j and b_j refer to the head and body of clause j.

$$\mathbf{A}[\![P]\!][\![a]\!]\Phi = \bigcup_{j=1}^{n} den\_atom[\![P]\!][\![a]\!][\![h_j]\!][\![b_j]\!]\Phi$$

We now define the meaning of an atom with respect to a single clause j in the program. Let h :- b be the renamed version of clause h_j :- b_j, let $\theta \in \Phi$ be a call substitution and θ_{entry} be the most general unifier of the goal θa and head h. If $\theta_{entry} = fail$, then the body of j^{th} clause does not contribute to the success substitutions of a. Otherwise, let $\Phi' = \mathbf{B}[\![P]\!][\![b]\!]\{\theta_{entry}\}$ be the exit substitutions of the body of j^{th} clause corresponding to the entry substitution θ_{entry}. The composition of each $\theta_{exit} \in \Phi'$ with θ gives us the set of success substitutions corresponding to one call substitution $\theta \in \Phi$ and union of this set over all $\theta \in \Phi$ results in the set of success substitutions of goal a. Thus, the denotation of an atom a (with respect to a single clause) maps a set of call substitutions to a set of success substitutions. The substitution $\theta_{exit} \circ \theta$ can be restricted to variables not occurring in clause j without affecting the correctness of the valuation functions. In order to keep the presentation simple, we omit the restriction step.

```
$den\_atom[\![P]\!][\![a]\!][\![h_j]\!][\![b_j]\!]\Phi =$
    let $(h, b) = rename(h_j, b_j)$ in
    $\bigcup_{\theta \in \Phi}$ {   let $\theta_{entry} = mgu(\theta a, h)$ in
            if $\theta_{entry} = $ fail then {}
            else $\{\theta_{exit} \circ \theta \mid \theta_{exit} \in \mathbf{B}[\![P]\!][\![b]\!]\{\theta_{entry}\}\}$
        }
```

Goal b_0 is just a conjunct of literals and its denotation is also a function that maps a set of substitutions to a set of substitutions. However, the initial set of substitutions for b_0 is the singleton set of identity substitution $\{\varepsilon\}$. The meaning of a program is simply the answer substitutions of the initial goal b_0 and is defined by the following equation.

$$\mathbf{P}[\![P]\!] \;=\; \mathbf{B}[\![P]\!][\![b_0]\!]\{\varepsilon\} \text{ where } P = h_1 \leftarrow b_1, \ldots, h_n \leftarrow b_n, \leftarrow b_0$$

3.4 An Example

We will derive the meaning of the familiar append program in standard semantics. Part of the derivation is shown in figure 1. Symbols starting with an upper case letter denote variables. Predicate, function and constant symbols start with a lower case letter.

```
append([], R, R).                                (1)
append([X|Y], Z, [X|W]) :-                       (2)
        append(Y, Z, W).
:- append([a], [b,c,d], A).                      (3)
```

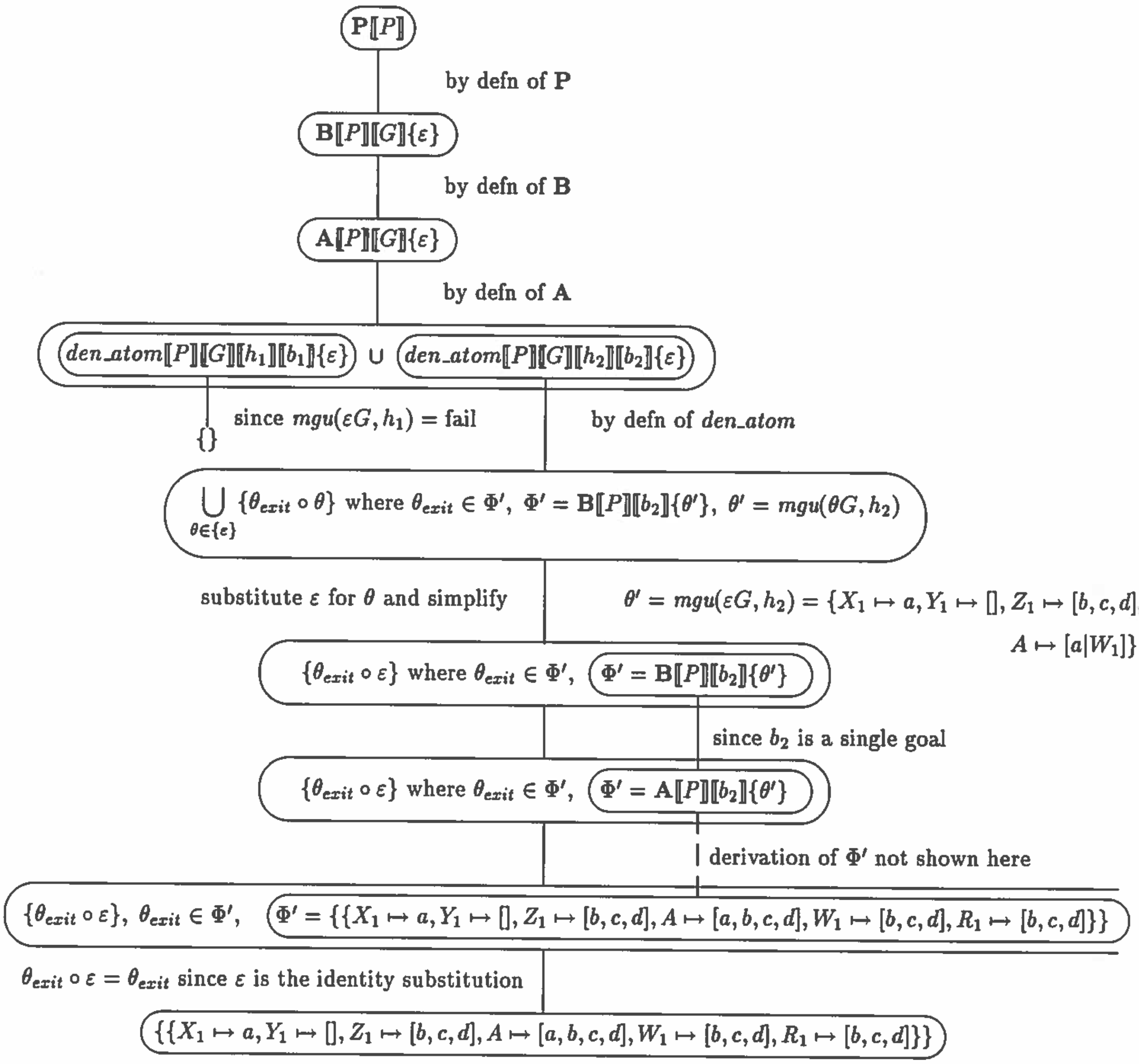
$\mathbf{P}[\![P]\!]$
by defn of **P**
$\mathbf{B}[\![P]\!][\![G]\!]\{\varepsilon\}$
by defn of **B**
$\mathbf{A}[\![P]\!][\![G]\!]\{\varepsilon\}$
by defn of **A**
$den\_atom[\![P]\!][\![G]\!][\![h_1]\!][\![b_1]\!]\{\varepsilon\} \cup den\_atom[\![P]\!][\![G]\!][\![h_2]\!][\![b_2]\!]\{\varepsilon\}$
$\{\}$
since $mgu(\varepsilon G, h_1) = \text{fail}$
by defn of *den_atom*
$\bigcup_{\theta\in\{\varepsilon\}} \{\theta_{exit} \circ \theta\}$ where $\theta_{exit} \in \Phi'$, $\Phi' = \mathbf{B}[\![P]\!][\![b_2]\!]\{\theta'\}$, $\theta' = mgu(\theta G, h_2)$
substitute ε for θ and simplify
$\theta' = mgu(\varepsilon G, h_2) = \{X_1 \mapsto a, Y_1 \mapsto [], Z_1 \mapsto [b,c,d]$
$A \mapsto [a|W_1]\}$
$\{\theta_{exit} \circ \varepsilon\}$ where $\theta_{exit} \in \Phi'$, $\Phi' = \mathbf{B}[\![P]\!][\![b_2]\!]\{\theta'\}$
since b_2 is a single goal
$\{\theta_{exit} \circ \varepsilon\}$ where $\theta_{exit} \in \Phi'$, $\Phi' = \mathbf{A}[\![P]\!][\![b_2]\!]\{\theta'\}$
derivation of Φ' not shown here
$\{\theta_{exit} \circ \varepsilon\}$, $\theta_{exit} \in \Phi'$, $\Phi' = \{\{X_1 \mapsto a, Y_1 \mapsto [], Z_1 \mapsto [b,c,d], A \mapsto [a,b,c,d], W_1 \mapsto [b,c,d], R_1 \mapsto [b,c,d]\}\}$
$\theta_{exit} \circ \varepsilon = \theta_{exit}$ since ε is the identity substitution
$\{\{X_1 \mapsto a, Y_1 \mapsto [], Z_1 \mapsto [b,c,d], A \mapsto [a,b,c,d], W_1 \mapsto [b,c,d], R_1 \mapsto [b,c,d]\}\}$

Figure 1: Derivation of $\mathbf{P}[\![P]\!]$

In the following derivations, P refers to the above program, G refers to the goal, ε the empty substitution. Let $append([X_1|Y_1], Z_1, [X_1|W_1])\ :-\ append(Y_1, Z_1, W_1)$ be a renaming of clause 2 and h_2 and b_2 its head and body. At each step of the derivation we either apply the definition of a semantic function or simplify one or more sub-expressions occurring in the previous step. Sub-derivations are indented for clarity.

- The meaning of the program P, i.e., $\mathbf{P}[\![P]\!] = \mathbf{B}[\![P]\!][\![G]\!]\{\varepsilon\}$ by definition of **P**.
- $\mathbf{B}[\![P]\!][\![G]\!]\{\varepsilon\} = \mathbf{A}[\![P]\!][\![G]\!]\{\varepsilon\}$ by definition of **B**.
- $\mathbf{A}[\![P]\!][\![G]\!]\{\varepsilon\} = den\_atom[\![P]\!][\![G]\!][\![h_1]\!][\![b_1]\!]\{\varepsilon\}\ \cup\ den\_atom[\![P]\!][\![G]\!][\![h_2]\!][\![b_2]\!]\{\varepsilon\}$ by definition of **A**.
- which is reduced to $den\_atom[\![P]\!][\![G]\!][\![h_2]\!][\![b_2]\!]\{\varepsilon\}$, since goal G does not unify with the head of clause one and hence $den\_atom[\![P]\!][\![G]\!][\![h_1]\!][\![b_1]\!]\{\varepsilon\} = \{\}$.
- Next we simplify $den\_atom[\![P]\!][\![G]\!][\![h_2]\!][\![b_2]\!]\{\varepsilon\}$ using he definition of *den_atom*. The entry substitution $\theta' = mgu(\varepsilon G, h_2) = \{X_1 \mapsto a, Y_1 \mapsto [], Z_1 \mapsto [b,c,d], A \mapsto [a|W_1]\}$. Using the entry substitution θ', we compute $\mathbf{B}[\![P]\!][\![b_2]\!]\{\theta'\}$ in the next two steps.
  - $\mathbf{B}[\![P]\!][\![b_2]\!]\{\theta'\} = \mathbf{A}[\![P]\!][\![b_2]\!]\{\theta'\}$, by definition of **B**. Applying the definition of **A** and simplifying, we get

    $\mathbf{A}[\![P]\!][\![b_2]\!]\{\theta'\} = den\_atom[\![P]\!][\![b_2]\!][\![h_1]\!][\![b_1]\!]\{\theta'\}\ \cup\ den\_atom[\![P]\!][\![b_2]\!][\![h_2]\!][\![b_2]\!]\{\theta'\}$
  - $den\_atom[\![P]\!][\![G1]\!][\![h_2]\!][\![b_2]\!]\{\theta'\}$ reduces to $\{\}$, since $\theta' b_2$ does not unify with the head h_2 of clause 2. $den\_atom[\![P]\!][\![b_2]\!][\![h_1]\!][\![b_1]\!]\{\theta'\}$ induces an entry substitution $\theta'' = \{W_1 \mapsto [b,c,d], R_1 \mapsto [b,c,d]\}$ for clause one and since the body of clause one is empty, the set of exit substitutions of clause one is $\{\theta''\}$. The success substitutions for clause one are obtained by composing θ'' with θ'. Thus, $den\_atom[\![P]\!][\![b_2]\!][\![h_2]\!][\![b_2]\!]\{\theta'\} = \Phi'$, where

    $\Phi' = \{\{\mathrm{X}_1 \mapsto \mathrm{a}, \mathrm{Y}_1 \mapsto [], \mathrm{Z}_1 \mapsto [\mathrm{b,c,d}], \mathrm{A} \mapsto [\mathrm{a,b,c,d}], \mathrm{W}_1 \mapsto [\mathrm{b,c,d}], \mathrm{R}_1 \mapsto [\mathrm{b,c,d}]\}\}$.
- Thus $\mathbf{B}[\![P]\!][\![b_2]\!]\{\theta'\} = \Phi'$. The final step is to compose each substitution in Φ' with ε. Since ε is the identity substitution, the result is Φ' itself, which is the set of success substitutions for goal G. If we restrict each substitution in Φ' to variables appearing in the goal (which is what is observable in an implementation), then the set of success substitutions for the goal is $\{\{A \mapsto [a,b,c,d]\}\}$.

To summarize, we defined a denotational semantics for a simple logic programming language with a left-to-right selection rule and a parallel search rule. We also derived the meaning of a small program. The purpose of using a denotational definition instead of an operational one is to show the relationship between the standard semantics and an abstract interpretation to be presented in the following section.

4 Abstract Interpretation

Abstract Interpretation can be thought of as performing non-standard computations in non-standard domains. The domains and computations are not arbitrary but bear some relationship to the standard domains and standard computation. We illustrate this with a very simple example. Suppose we want to know the sign of the result of simplifying the expression 984×-32. We don't perform actual multiplication to find the answer. The rule of signs tells us that the result will be negative. We mapped the standard domain of computation, integers, onto the set $\mathrm{D} = \{\mathrm{zero}, \mathrm{pos}, \mathrm{neg}\}$ and replaced the multiplication operation on the integers with another operation (the rule of signs) $f : \mathrm{D} \times \mathrm{D} \rightarrow \mathrm{D}$.

Cousot and Cousot [7] formalized the idea that static analyses of programs can be viewed in terms of abstract interpretation. Their work was concerned with the analysis of flow chart programs. A flowchart program is represented as a graph. Nodes correspond to basic operations such as assignment and test. Arcs represent control flow. Certain constraints are placed on the nodes. For example, a test node must have a single predecessor node and exactly two successor nodes corresponding to the true and false branches. The state of a program is represented by a program counter and an environment which associated values with identifiers. Semantic functions define the meaning of various operations as transformations on the state. Cousot and Cousot define a *static semantics* (later on called *collecting semantics*) that associates with each program point (arc) the set of all environments that may prevail at that point during any execution of the program. Properties of programs are then derived from finitely computable approximations of these sets.

For logic programs, the natural counterpart to states is substitutions. Literals correspond to nodes and a program point connects two literals in the body of a clause. The collecting interpretation or collecting semantics of a program associates with each program point the set of all substitutions that *may prevail* at that point during an execution of the goal(s). All analyses are done using abstract interpretations that are finitely computable approximations of these sets of substitutions.

The semantics based approach to analysis is to define a core semantics for the language that leaves some domains and semantic functions unspecified. An interpretation I of the core semantics supplies the missing domains and semantic functions. The word *interpretation* is used to emphasize that the undefined function symbols and domains are *interpreted* to suit our purpose (subject to some conditions). The standard semantics and collecting semantics are shown to be interpretations of the core semantics. The collecting semantics is almost the same as the standard semantics except that it usually provides more information; it preserves the association between program points and the sets of substitutions encountered at these points during execution.

An interpretation I of the core semantics is an abstract interpretation if the the domains and semantic functions of I are abstractions of their counterparts in the collecting semantics and if I safely models the collecting semantics.

We now develop a core semantics for the language defined in section 3; show that the standard semantics of section 3 can be seen as an interpretation of the core semantics;