# A Minimal Function Graph Semantics for Logic Programs

by

Will Winsborough

Computer Sciences Technical Report #711
August 1987

# A Minimal Function Graph Semantics for Logic Programs

Will Winsborough

August 14, 1987

**Abstract**

The problem is to devise an abstract interpretation framework for static analysis of logic programs. We seek a framework in which to couch analyses that infer stronger assertions than do previous efforts, but that remain plausibly efficient. The framework offered here is designed to efficiently accommodate large abstract domains that introduce imprecision only as diversity naturally arises in the inferences.

Our semantics is based on the view of predicates as partial functions from sets of activation instances to sets of result instances. It is essentially a *minimum function graph (mfg)* semantics, following [8], though it treats sets of activations rather than individuals because doing so allows us to use large abstract domains, and hence infer stronger properties, without sacrificing efficiency.

First we construct a function whose fixpoint defines a *total function graph (tfg)* semantics. We then define the *mfg* semantics as the *tfg* restricted to the call instances reachable from some specified query set.

Finally, we consider how to compute the *mfg* directly, without having to compute the (large) *tfg*. If reachability is added to the iteration function whose fixpoint defines the *tfg*, problems arise because we treat sets of activations. The function is no longer monotonic, and its (transfinite) Kleene's sequence doesn't converge to the *mfg*. The Kleene's sequence does converge, however, and we define our *partial function graph (pfg)* semantics as its limit.

For finite abstract domains the *pfg* semantics is computable and is generally much smaller and much less expensive to compute than the *tfg*. In the main technical result of the paper, we prove the *pfg* agrees exactly with the *mfg*, except that a few activation sets (though, *not* individuals) erroneously appear reachable. The difference has no meaning for most execution models. So, despite its theoretical unpleasantness, the *pfg* is very practical.

## 1 Introduction

A data flow analysis is an attempt to automatically infer properties of program executions with a terminating procedure. Termination of the procedure can be guaranteed for some properties that are weaker than the exact details of the program execution. The technique of abstract interpretation, based on fixpoint semantics constructions, provides a framework for constructing data flow analyses so that their correctness can be easily verified. Two essentially homomorphic interpretations of a programming language are formally related, one precise (called the concrete interpretation), the other approximate (called an abstract interpretation). The concrete interpretation induces a collecting semantics that agrees precisely with the standard semantics. An abstract interpretation induces an approximate semantics that is a data flow analysis. Given that the interpretations satisfy some relatively simple properties, powerful fixpoint theorems can be invoked to establish that the abstract semantics is a (weaker) implication of the concrete semantics.

We follow [7,8] and others by constructing a *core semantics* using a collection of domains and auxiliary functions, called an interpretation, that are assumed to have certain properties, and that act as formal parameters to the construction. Different actual interpretations induce different semantics. In this paper we construct a core semantics for pure Horn Clause logic programs and give an interpretation that induces a collecting semantics that agrees with SLD-resolution. Abstract interpretations can be provided to generate various data flow analyses.

To demonstrate that an abstract interpretation generates a safe approximate semantics, there must be an abstraction function that maps concrete domain elements to abstract domain elements and that is (almost) a homomorphism relative to the iteration functions whose least fixpoints define the respective semantics. When a core semantics is given, the homomorphism requirement is factored into requirements of the interpretations' auxiliary functions. The result of the (almost) homomorphism is that the abstract semantics is implied by the concrete semantics. This fact comes from well known theorems about fixpoints[3]. Because this concept is well understood and because of space considerations, we do not include a formal treatment of safeness of abstract interpretations for our framework.

### 1.1 Our Motive

This work is of general interest because it gives an abstract interpretation framework for a declarative language based on a function domain that takes sets of activation instances to sets of result instances, but that does not use power domains. Further, it defines a *minimal function graph (mfg)* semantics in such a domain and develops a practical way to compute it for finite, though possibly large, domains.

Our particular interest is to design a framework in which to couch analyses that improve the strength of the assertions inferred over the analyses that have been proposed thus far in the literature. We want a framework that will allow such an improvement while keeping overhead at a plausible level. Thus, the framework presented here is designed to accommodate large abstract domains that can represent properties with great precision, but that introduce imprecision as diversity arises in inferences. We discuss the issues involved in that accommodation in Section 1.4.

Data flow analyses of logic programs include mode inference, which infers variable instantiation patterns at call activation and result[11,6,4,5], and occur check avoidance, which finds calls in which there is no threat that unification will introduce cycles into structures[13,14].

We are primarily interested in mode inference, which is useful in sequential compilers for optimizing unification and storage management[15], and to parallel execution models, where we may execute concurrently two literals that are known to share no free variables[2,1].

The best published work on mode inference has been done by Debray. Mellish has also proposed an analysis for mode inference [11], but unfortunately that work fails to treat correctly the problem of variable aliases. The method of Debray, developed in [6,4,5], treats predicates as instantiation state transforms. A result instantiation description is associated with each activation instantiation description that could arise while executing the program. Debray's analysis is an example of the kind of analysis our framework can be used to verify. However, we have constructed the framework with larger abstract domains in mind.

Using our framework, we have developed and implemented an abstract interpretation that handles mode inference and that infers stronger properties than did previous analyses for mode inference. That will be the subject of a forthcoming paper.

## 1.2 Possible Frameworks

The original exposition of abstract interpretation by Cousot and Cousot gave a collecting semantics for a simple imperative language without procedure calls[3]. That semantics constructs simultaneously a description of the execution states possible at each point in the program. In their language, a program is represented by a directed graph with a node for each instruction and a directed arc from each node to the node for each possible next instruction. The arcs are the "program points" where execution state descriptions are collected. Cousot and Cousot call an execution state description a context, and a vector of these descriptions, one for each program point, a context vector.

In the Cousot framework, an iteration function (called an interpretation function in [3]) simulates one step of program execution simultaneously at all nodes in the graph. This iteration function takes one context vector and produces another. It simulates the operation at each node by looking at the possible incoming states according to the information in the input context vector and generating a description of the possible outgoing states on that basis. The program's collecting semantics is the least fixpoint of this iteration function.

The iteration function for a logic program will also take an input context vector. To produce an output context vector it will simulate the execution of each clause by simulating its calls according to the input context vector. The unification of calls with call results and the composition of call solutions according to the clause are the two basic operations of logic programs. Both of these are performed to complete one application of the iteration function. The context vector should represent the results of various calls. Additionally, we would like to determine which call activations are reachable from some specified query set. But what form should the context vector take?

C. S. Mellish has constructed an abstract interpretation framework for logic programs [12]. In it, procedure entry and exit are taken as the program points. Canonical literals, normalized representatives of their renaming equivalence class, are collected into two sets, called Input and Output. Input (resp. Output) contains call activation (resp. result) instances that could possibly occur during the execution of one of the programmer specified queries. In that framework, the two sets constitute the context vector. This approach does not preserve information about which result instances correspond to which activation instances. Because the semantic construction must work for either the concrete or the abstract interpretation, one cannot rely on unification to screen out inappropriate results. All result instances that have been found to occur for a given predicate are treated as possible results for every call to that predicate. Irrelevant results propagate and accumulate.

In search of a more powerful framework, we observe that (pure) logic programs are referentially transparent; the behavior of a call is entirely determined by the activation instance. If we consider *sets* of activation instances as the program points, it is possible to get a finite approximation domain, and hence termination, by selecting representative sets to constitute the approximation domain.

Thus we conclude that the context vector shall be a function from activation sets to result sets. Frequently we shall say "description" rather than "set" because an abstract domain may be any homomorphic structure. We understand the term "minimal function graph", barrowed from [8], to suggest a convenient representation for the function.

## 1.3 A Minimal Function Graph Framework

As we shall see, it is a relatively straightforward matter to define **TFG**[[Pr]], a *total function graph (tfg)* semantics for pure Horn Clause programs. It is a function that takes any set of calls to some predicate and returns the set of result instances and is defined in Section 2.4. Under the concrete interpretation, the *tfg* semantics agrees exactly with a standard operational semantics that evaluates a query by computing *all* solutions to the first call, then, for each of those solutions, computing all solutions to the corresponding substitution instance of the second call, and so on. Such a model could be made realistic by using eager consumers to avoid problems with infinite solution sets and other non-terminating calls. That execution model is complete and is the basis for the parallel execution model of Kalé[9]. However, for our purposes it is clearer to ignore termination problems and adopt the more declarative view of the model as a composition of operations on sets. We will refer to this loosely as the *all-solutions* model.

The *tfg* semantics does not include any information about which activation sets are reachable. A *minimal function graph (mfg)* semantics is a function that is only defined on those activation sets that are reachable from a given query set. Obviously, this *mfg* semantics can be constructed from the *tfg* by constructing the set of reachable call activation sets and restricting the *tfg* to that. Thus, the *mfg* contains not only information about call results, but also about which calls are reachable. The latter information is represented by the domain over which the function is defined.

Unfortunately, we do not know how to construct the *mfg* with a single least fixpoint operation. (Reachability requires a fixpoint operation.) If one incorporates reachability of activations into the *tfg* iteration function, the function becomes non-monotonic. Which activation instance sets appear reachable at some clause body call depends on the partial results currently reported for other calls to the left in the clause. For example, suppose we are trying to analyze the following program in the concrete interpretation.

```
p(X,Y) :- s(X), r(X,Y).
s(a).
s(X) :- t(X).
t(b).
r(a,a).
r(b,b).
?- p(X,Y).
```

Further suppose we have inferred that $\{s(X)\} \mapsto \{s(a)\}$, but have not yet inferred that s(b) is also a solution to s(X). At this stage it appears that $\{r(a,Y)\}$ is a reachable activation set. If we repeatedly apply the iteration function, $\{r(a,Y), r(b,Y)\}$ will eventually be inferred to be reachable. But the *mfg* only includes the larger set, since the execution model collects *all* solutions to s(X) before trying r(X,Y).

As we will show, a limit to the Kleene's sequence[1] still exists, defining what we call here the *partial function graph (pfg)* semantics. But it is not the *mfg* because some subsets of genuinely reachable activation sets incorrectly appear reachable.

[1]The Kleene's sequence of the function $f : A \rightarrow A$ is defined to be the transfinite sequence $\{f^{\gamma}(\bot)\}_{\gamma}$, where $f^0(x) =_{df} x$; $f^{\delta+1}(x) =_{df} f(f^{\delta}(x))$; and $f^{\alpha}(x) =_{df} \sqcup\{f^{\delta}(x) | \delta \sqsubset \alpha\}$, for $\alpha$ a limit ordinal.

Of course, most applications are for execution models in which only the reachable *individual* activations matter. On which individuals these are the *mfg* and the *pfg* agree. Furthermore, in Section 3.2 we prove that on the reachable activation descriptions the *mfg*, the *pfg*, and the *tfg* are all equal.

The other difference between *pfg* and *mfg* is that the iteration function for the *pfg* is not continuous, so the sequence whose limit defines the *pfg* will not in general converge by stage $\omega$. However, termination is ensured when a finite approximation domain is used. And this restriction turns out to be essentially necessary if the *mfg* is to consist of finitely many pairs.

## 1.4 Recent Related Work

We have recently become aware of closely related work by N. Jones and H. Søndergaard, reported in [7]. Roughly speaking, that framework is based on functions from substitutions to sets of substitutions. We believe that the choice of substitutions versus canonical term-tuples is not significant.

The Jones-Søndergaard formulation is very elegant. The domains for activations and results are distinct in the core semantics and the former is a subset of the latter. In the concrete interpretation, the activations are singletons of substitutions and the results are arbitrary sets of substitutions. However, the two domains may be the same actual domain in an abstract interpretation.

Suppose $B_1,\ldots,B_m$ is a clause body. The Jones-Søndergaard core semantics specifies that, when simulating $B_i$, the call must be evaluated under each activation domain element less than the result domain element describing the set of substitutions resulting from $B_1,\ldots,B_{i-1}$. So, in the concrete interpretation, each member of the set of substitutions resulting from $B_1,\ldots,B_{i-1}$ is applied to $B_i$ in turn. If the abstract interpretation makes the activation domain and the result domain the same, then there is only one instance of $B_i$ to evaluate. But we are interested in accommodating large abstract domains where it is possible to preserve precision until diversity arises in the inference. Such domains are relatively tall. Using such a domain, it may be necessary in the Jones-Søndergaard semantics to cycle through a large number of call activations. Our framework allays this threat to efficiency by letting the granularity of activation description for which results are collected be determined by the amount of diversity in each activation set inferred to be reachable. Rather than require the abstract interpretation to fix the granularity of activation description for which results are collected, our method allows that collecting granularity to be determined by the granularity of the activation descriptions reachable from the query in our all-solutions model.

Another feature of the Jones-Søndergaard framework that differs significantly from ours is how reachability inferences are collected. We require that a query set be provided. It provides the basis for reachability, and, for the most part, results are only collected for reachable activation descriptions. The Jones-Søndergaard method collects a result for *every* activation, and in the process, collects a log of activations that may be reachable from each of those activations.

From a theoretical standpoint, we admire the Jones-Søndergaard construction very much. From a practical standpoint, our method appears preferable, particularly for use with large abstract domains.

## 1.5 Organization

The rest of this paper is organized as follows. In Section 2, we construct the *mfg* semantics and give the concrete interpretation for the formal parameters to the construction. Then, in Section 3, we present the *pfg* construction, show that it is well defined and prove that it agrees with the *mfg* wherever the latter is defined.

Because of space considerations we do not present a proof that the concrete semantics agrees with the SLD-resolution semantics.

We assume the reader is familiar with the basic ideas of abstract interpretation[3] and of logic programming[10]. Familiarity with [8] would help.

# 2 The MFG Construction

This section gives the core semantics construction and presents the concrete interpretation. The core semantics construction defines the *tfg*, the set of reachable activation descriptions, and the *mfg*.

## 2.1 Syntactic Domains

Let `Func` denote the set of $\ell$-placed function symbols, $\ell \geq 0$. Let `Var` denote a countably infinite set of variables. Let `Term` denote the set of logical terms over `Func` and `Var`. Let `Pred` denote the set of $k$-placed predicate symbols. We assume for simplicity of presentation that one fixed $k$ holds for all predicates in any given program. Let `Literal` denote the set of logical literals over `Pred` and `Term`.

Let `Clause` be the set of Horn clauses. Henceforth, all clauses mentioned are implicitly Horn. We write clauses `H :- B`$_1$,...,`B`$_m$, $m \geq 0$. As usual, the ':-' is omitted when $n = 0$. `H` is called the clause head; together, `B`$_1$,...,`B`$_m$ make up the clause body. Let `Prog` $=_{df}$ {`Pr`|`Pr` $\subset$ `Clause` and `Pr` is finite} be the program domain. `Pr` $\in$ `Prog` is a collection of Horn clauses constituting a given program. We assume that the number of (non-primative) predicates occurring in `Pr` is $n$ and we refer to them as $p_1, p_2, \ldots, p_n$. A clause for predicate $i$ is one that has $p_i$ in its head. `Queries` $\in \wp$(`Literal`) will denote the set of possible queries associated with `Pr`. Without loss of generality, we make the simplifying assumption that elements of `Queries` are single negative literals.

## 2.2 Interpretations

The construction of our semantics is parameterized by two domains and several auxiliary functions. These parameters, taken together, constitute an interpretation[8]. Basic assumptions are required of an interpretation for the construction to be well defined.

**Definition.** An interpretation consists of the following:
Two Lattices:

> $\mathsf{Arg}$, $\top_{\mathsf{Arg}}$, $\bot_{\mathsf{Arg}}$, $\sqcup_{\mathsf{Arg}}$, $\sqsubseteq_{\mathsf{Arg}}$
> $\mathsf{CArg}$, $\top_{\mathsf{CArg}}$, $\bot_{\mathsf{CArg}}$, $\sqcup_{\mathsf{CArg}}$, $\sqsubseteq_{\mathsf{CArg}}$
> $\mathsf{Env}$, $\top_{\mathsf{Env}}$, $\bot_{\mathsf{Env}}$, $\sqcup_{\mathsf{Env}}$, $\sqsubseteq_{\mathsf{Env}}$

Monotone Functions:

> $\mathsf{Apply} : [0..m] \times \mathsf{Env} \times \mathsf{CArg} \times \mathtt{Clause} \longrightarrow \mathsf{Env}$
>> Apply must be bottom-preserving and continuous
>> in its second and third arguments.
>
> $\mathsf{Project} : [0..m] \times \mathsf{Env} \times \mathtt{Clause} \longrightarrow \mathsf{Arg}$
>> Project must be bottom-preserving and continuous
>> in its second argument.
>
> $\mathsf{GetArg} : \wp(\mathtt{Literal}) \longrightarrow \mathsf{Arg}^n$
> $\mathsf{GetEnv} : \mathtt{Clause} \longrightarrow \mathsf{Env}$
> $\mathsf{Canon} : \mathsf{Arg} \longrightarrow \mathsf{CArg}$
> $\mathsf{FreeInst} : \mathsf{CArg} \longrightarrow \mathsf{Arg}$

$x \in \mathsf{Arg}$ represents a set of literal instances. $x$ does not indicate which predicate symbol occurs in the literals, just which $k$-tuple of terms appear as arguments. In the sequel, elements of Arg will simply be said to represent sets of instances.

The elements of CArg represent sets of instances that are canonical representatives of their variable renaming equivalence class. So, $x \in \mathsf{Arg}$ can contain two instances that are the same up to variable renaming, but, $y \in \mathsf{CArg}$ cannot.

The context vector, which our iteration function operates on, will be a vector of functions from CArg to CArg. Recall that the iteration function takes an *input context vector* and produces an *output context vector* by simulating each clause execution based on the call results given by the input context vector. Each component of the vector will correspond to a different predicate $p_i$, and in the fixpoint will represent the behavior of sets of calls to $p_i$.

An element of Env represents a *local clause context.* The local clause context is local in the sense that it's lifetime is the duration of the simulation of a single clause execution. It is used by the iteration function for recording intermediate results. But the output context vector only records what the local clause context has to say about the state of the clause *head* after clause execution simulation.

Initially, a clause's local context represents only one instance of the clause: the one that appears in the program text. But when the clause is activated with a set of call instances for the head predicate, the local context may then represent many clause instances. Each call in the clause is then simulated based on the function values in the appropriate component of the input context vector, updating the local context accordingly. This phase may also change the number of clause instances represented by the local context, as some call instances fail and others produce multiple results. When all the calls have been simulated, the head of each clause instance represented by the local context is one possible result of the set of activation instances. An element of CArg, representing a set of result instances, is specified by the head under the local clause context.

For a given $x \in$ CArg representing a set of instances, and for a given predicate symbol $p_i$, the $i$'th component of the output context vector will be a function whose value at $x$ is $y \in$ CArg where $y$ is the meet ($\sqcup_{\mathsf{CArg}}$) of the results found using each clause for $p_i$ as described in the last paragraph. In the concrete interpretation, the meet is set union. In abstract an interpretation the meet may represent a larger set than the union.

Members of Env are not required to specify the predicate symbols for the literals that make up the clause. It only must represent the possible substitutions or clause instances that are in force.

Apply simulates the effect on the local context of unifying one literal in the clause with a set of instances. The integer parameter indicates which clause literal is being unified. Apply is used both to activate the clause and to simulate call execution. In the former case, Apply simulates unification of the clause head with the set of activation instances, an element of Arg, whose execution the iteration function is simulating. In the latter, Apply is used by the iteration function to simulate the execution of each call in turn. For a call to $p_i$, the iteration function evaluates the $i$-th component of the input context vector on the set of activation instances specified in the local context. The result is a set of instances that is supplied to Apply. Apply simulates unification of each activation instance (from the local context) with each result instance (from the input context vector), composing appropriately the successful unifiers in the local clause context that is Apply's result.

In the abstract interpretation Apply will not in general consider each pair of instances separately. The role of the abstract interpretation is to lump many instances together so they can be regarded collectively.

Project is used to get the set of instances for a particular clause literal specified by the local clause environment. That is needed to be able to evaluate the components of the input context vector, and to get the the head instances from the local clause context when all the calls have been simulated.

GetArg generates, as a description of a set of queries, a tuple of representations of sets of instances, one for each predicate in the program. Each component represents, for the corresponding predicate, the set of activation instances in the set of queries. Recall that we make the simplifying assumption that each query contains only one literal.

GetEnv generates a local context for the given clause.

Canon and FreeInst go back and forth from Arg to the equivalent CArg. We remark on them further in Section 2.5.

Now we begin constructing our semantics.

## 2.3 The Semantic Domain

The collecting semantics of Pr∈Prog is an $n$-placed vector of transforms, **TFG**[[Pr]], where $n$ is the number of predicates defined in Pr. The $i$'th predicate's semantics is given by $\mathbf{TFG}[[\mathtt{Pr}]]_i$.[2]

**TFG**[[Pr]] will be constructed in the next section. We now construct and name the transform space.

> $\mathbf{Trans} =_{df} \mathsf{CArg} \rightarrow \mathsf{CArg}$ is the domain of predicate denotations.
>
> $\mathbf{Trans}^n$ is the domain of predicate environments or context vectors.[3]

**Trans** is a complete lattice with[4]

$$\top_{\mathbf{Trans}} =_{df} \lambda x{:}\mathsf{CArg}\ .\ \top_{\mathsf{CArg}}$$
$$\bot_{\mathbf{Trans}} =_{df} \lambda x{:}\mathsf{CArg}\ .\ \bot_{\mathsf{CArg}}$$
$$f \sqsubseteq_{\mathbf{Trans}} g \ \ \text{if}_{df}\ \forall x \in \mathsf{CArg}\ .\ f(x) \sqsubseteq_{\mathsf{CArg}} g(x)$$
$$f \sqcup_{\mathbf{Trans}} g \ =_{df} \lambda x{:}\mathsf{CArg}\ .\ f(x) \sqcup_{\mathsf{CArg}} g(x)$$

So $\mathbf{Trans}^n$ is also a complete lattice with join and order defined pointwise.

## 2.4 The Total Function Graph Construction

Now we construct the iteration function, $\mathbf{Iterate_{TFG}}$, whose fixpoint defines the program *tfg* semantics. In the definitions we use the functions *vector*, *fix* and *index*. *vector* takes a function with domain [1..n] and returns the n-placed vector whose $i$'th component is the value of the function applied to $i$. *fix* is the least fixpoint operator. *index* takes a literal and gives back the integer in [1..n] that identifies its predicate symbol. It is used to select the appropriate component of the context vector.

The *tfg* semantics of Pr ∈ Prog are given by

> $\mathbf{TFG}$: Prog $\rightarrow \mathbf{Trans}^n$

[2] Subscripting denotes component selection.

[3] The superscript denotes standard cross product.

[4] In these typed lambda expressions the colon separates the formal parameter from its domain. As usual, the dot separates the parameter list from the expression body. In the universally quantified sentence (third line) the dot is used to separate the quantification of the variable from the rest of the formula.

$$\mathbf{TFG}[[\texttt{Pr}]] \ =_{df}\ \mathit{fix}\ \lambda\phi{:}\mathbf{Trans}^n.\ \mathbf{Iterate}_{\mathbf{TFG}}[[\texttt{Pr}]](\phi)$$

where **Iterate** is defined by

$$\begin{aligned}
&\mathbf{Iterate}{:}\ \texttt{Prog} \to \mathbf{Trans}^n \to \mathbf{Trans}^n\\
&\mathbf{Iterate}[[\texttt{Pr}]](\phi)\ =_{df}\ \mathit{vector}\Big(\lambda i{:}[1..n]\ .\ \lambda x{:}\mathsf{CArg}\ .\\
&\quad \bigsqcup_{\mathsf{CArg}} \Big\{\ \mathbf{UseClause}[[\texttt{H :- B}_1,\ldots,\texttt{B}_m]](x,\phi)\ |\\
&\qquad\qquad \texttt{H :- B}_1,\ldots,\texttt{B}_m \text{ is a clause for the } i\text{'th predicate in } \texttt{Pr}\Big\}\Big)
\end{aligned}$$

and where **UseClause** and **Trace** are defined by[5]

$$\begin{aligned}
&\mathbf{UseClause}{:}\ \texttt{Prog} \to \mathsf{CArg} \times \mathbf{Trans}^n \to \mathsf{CArg}\\
&\mathbf{UseClause}[[\texttt{H :- B}_1,\ldots,\texttt{B}_m]](x,\ \phi)\ =_{df}\\
&\quad \mathsf{Canon}(\mathsf{Project}(0,\mathbf{Trace}(\\
&\quad \mathsf{Apply}(0,\ \mathsf{GetEnv}[[\texttt{H :- B}_1,\ldots,\texttt{B}_m]],\ x,\ <\texttt{H :- B}_1,\ldots,\texttt{B}_m>),\\
&\quad <\texttt{H :- B}_1,\ldots,\texttt{B}_m>,\ 1,\ \phi),\ <\texttt{H :- B}_1,\ldots,\texttt{B}_m>))
\end{aligned}$$

$$\begin{aligned}
&\mathbf{Trace}{:}\ \mathsf{Env}\ \times \texttt{Clause}\ \times \mathbf{Z}^+ \times \mathbf{Trans}^n \to \mathsf{Env}\\
&\mathbf{Trace}(E,\ <\texttt{H :- B}_1,\ldots,\texttt{B}_m>,\ j,\ \phi)\ =_{df}\\
&\quad \mathit{if}\ j > m\ \mathit{then}\ E\\
&\quad \mathit{else}\ \mathbf{Trace}(\mathsf{Apply}(j,\ E,\\
&\qquad \mathsf{FreeInst}(\phi_{\mathit{index}(\texttt{B}_j)}(\mathsf{Canon}(\mathsf{Project}(j,\ E,\ <\texttt{H :- B}_1,\ldots,\texttt{B}_m>)))),\\
&\qquad <\texttt{H :- B}_1,\ldots,\texttt{B}_m>),\ <\texttt{H :- B}_1,\ldots,\texttt{B}_m>,\ j+1,\ \phi)
\end{aligned}$$

**UseClause** is continuous in $\phi$, since it looks at a finite number of $\phi$ values and is a finite composition of continuous functions. So, being pointwise continuous, $\mathbf{Iterate}_{\mathbf{TFG}}$ is also continuous.

## 2.5 The Concrete Interpretation

We construct the concrete interpretation. It will induce a semantics that agrees with the standard semantics.

The concrete interpretation is based on functions from sets of term-tuples to sets of term-tuples. If $x \in \mathsf{Arg}_{\mathsf{concrete}}$ then $\vec{t} \in x$ is a $k$-tuple of terms.

Let $\wp(S)$ denote the powerset of $S$. Then

$$\mathsf{Arg}_{\mathsf{concrete}}\ =_{df}\ \wp(\texttt{Term}^k)$$

is a complete lattice under

$$\begin{aligned}
&\sqsubseteq_{\mathsf{Arg}_{\mathsf{concrete}}}\ =_{df}\ \subseteq \text{ (the subset relation), with}\\
&\top_{\mathsf{Arg}_{\mathsf{concrete}}}\ =_{df}\ \texttt{Term}^k,\\
&\bot_{\mathsf{Arg}_{\mathsf{concrete}}}\ =_{df}\ \{\}\ \text{(the empty set), and}
\end{aligned}$$

[5] $\mathbf{Z}^+$ represents the positive integers.

$\sqcup_{\mathsf{Arg}_{\mathsf{concrete}}} =_{df} \cup$ (set union).

As we have discussed, the set `Term` is partitioned into variable renaming equivalence classes. Because the results of two calls in the same equivalence class are also in the same equivalence class, we select a representative of each class and only define semantics on the representatives.

Let *Normalize* : $\mathtt{Term}^k \to \mathtt{Term}^k$ return a canonical representative of its argument's equivalence class under variable renaming.[6] For example, *Normalize* may rename the variables using a prefix of some enumeration of `Var`, ordered by first occurrence in the argument. However, to avoid variable name conflicts, we make the requirement that the result of *Normalize* does not contain any variables used in the program text. This could be formalized by dividing the variable space into two infinite sets, one to be used in programs, the other to be used in normalized terms.

By *Normalize*($\mathtt{Term}^k$) we denote the function's range. Thus

$$\mathsf{CArg}_{\mathsf{concrete}} =_{df} \mathit{Normalize}(\mathtt{Term}^k)$$

is a complete lattice under

$\sqsubseteq_{\mathsf{CArg}_{\mathsf{concrete}}} =_{df} \sqsubseteq_{\mathsf{Arg}_{\mathsf{concrete}}}$, with
$\sqcup_{\mathsf{CArg}_{\mathsf{concrete}}} =_{df} \sqcup_{\mathsf{Arg}_{\mathsf{concrete}}}$,
$\top_{\mathsf{CArg}_{\mathsf{concrete}}} =_{df} \wp(\mathit{Normalize}(\mathtt{Term}^k))$, and
$\bot_{\mathsf{CArg}_{\mathsf{concrete}}} =_{df} \bot_{\mathsf{Arg}_{\mathsf{concrete}}}$.

Recall $\mathsf{E} \in \mathsf{Env}_{\mathsf{concrete}}$ represents a set of clause instances. For $\mathsf{E} \in \mathsf{Env}_{\mathsf{concrete}}$, and $e \in \mathsf{E}$, $e_i \in \mathsf{Arg}_{\mathsf{concrete}}$ denotes the term-tuple that is the $i$'th component of $e$, where $i \in [1..m+1]$, and $m$ is the number of body literals in the clause.

$\mathsf{Env}_{\mathsf{concrete}} =_{df} \bigcup_{m \in \mathrm{N}} \wp((\mathtt{Term}^k)^{\mathsf{m}+1})$
$\sqsubseteq_{\mathsf{Env}_{\mathsf{concrete}}} =_{df} \subseteq$
$\sqcup_{\mathsf{Env}_{\mathsf{concrete}}} =_{df} \cup$
$\top_{\mathsf{Env}_{\mathsf{concrete}}} =_{df} (\mathtt{Term}^k)^{\mathsf{m}}$
$\bot_{\mathsf{Env}_{\mathsf{concrete}}} =_{df} \{\}$

Next we define the functions in the interpretation.

$\mathsf{Apply} : [0..\mathsf{m}] \times \mathsf{Env} \times \mathsf{CArg} \times \mathtt{Clause} \to \mathsf{Env}$
$\mathsf{Apply}_{\mathsf{concrete}}(i, E, x, \texttt{<H :- B}_1,\ldots,\texttt{B}_m\texttt{>}) =_{df} \left\{\sigma(e) \mid e \in E, \vec{t} \in x, \text{ and } \sigma = \mathit{mfg}(e_i, \vec{t}) \text{ is not fail}\right\}$

$\mathsf{Project} : [0..\mathsf{m}] \times \mathsf{Env} \times \mathtt{Clause} \to \mathsf{Arg}$
$\mathsf{Project}_{\mathsf{concrete}}(i, E, \texttt{<H :- B}_1,\ldots,\texttt{B}_m\texttt{>}) =_{df} \left\{\vec{t} \mid e \in E \text{ and } \vec{t} = e_i\right\}$

$\mathsf{GetArg} : \wp(\mathtt{Literal}) \to \mathsf{Arg}$

[6]The scope of variables is, of course, the whole term-tuple.

$\mathsf{GetArg}_{\mathsf{concrete}}(\mathtt{Queries}) \;=_{df}\; vector\Big(\lambda i{:}[1..n]\;.$
$\quad\Big\{\vec{t} \mid \text{there is a literal } p_i(\vec{t}) \in \mathtt{Queries}.\Big\}\;\Big)$

$\mathsf{GetEnv} : \mathtt{Clause} \rightarrow \mathsf{Env}$
$\mathsf{GetEnv}_{\mathsf{concrete}}[[\mathtt{H} \;\text{:-}\; \mathtt{B}_1,\ldots,\mathtt{B}_m]] \;=_{df}\; \{e\},$
$\quad$ where $e_1$ is the tuple of arguments to $\mathtt{H}$
$\quad$ and $e_i$ is the tuple of arguments to $\mathtt{B}_{i-1}$, for $i \in [2..m+1]$.

$\mathsf{Canon} : \mathsf{Arg} \rightarrow \mathsf{CArg}$
$\mathsf{Canon}_{\mathsf{concrete}}(x) \;=_{df}\; \Big\{y \mid \exists\vec{t} \in x \text{ and } y = Normalize(\vec{t})\Big\}$

$\mathsf{FreeInst} : \mathsf{CArg} \rightarrow \mathsf{Arg}$
$\mathsf{FreeInst}_{\mathsf{concrete}}(Y) \;=_{df}\; \Big\{\vec{t} \mid \exists y \in Y,\, y = Normalize(\vec{t}), \text{ and}$
$\quad$ the variables in $y$ have not been used yet in the computation.$\Big\}$

Although $\mathsf{Apply}_{\mathsf{concrete}}$ does not use its fourth argument, abstract interpretations may need to. The same is true of $\mathsf{Project}_{\mathsf{concrete}}$. In general, $\mathsf{E} \in \mathsf{Env}$ represents a set of substitutions.

$\mathsf{FreeInst}_{\mathsf{concrete}}$ is not really a function, since it returns different members of the renaming equivalence class each time it's called. But we decline to make the tedious avoidance of variable name conflict more formal. In all our constructions, any result from $\mathsf{FreeInst}_{\mathsf{concrete}}$ is put through $\mathsf{Canon}_{\mathsf{concrete}}$ in such a way as to make the final results well defined.

## 2.6 The Minimal Function Graph Construction

This section constructs the *mfg*, which is a partial function, so we must add partial functions to our semantic domain. This is accomplished by following Jones and Mycroft[8]. We augment the activation description domain with a new bottom element that is less than all other elements and that represents unreachable calls, that is, calls on which the *mfg* is not defined.

As an intermediate step toward constructing the *mfg*, we construct the set of reachable activation descriptions from the *tfg* and a user specified set of queries.

$\mathbf{CArg_{bot}} \;=_{df}\; \mathsf{CArg} \cup \{\mathbf{bot}\}$
$x \sqsubseteq_{\mathbf{CArg_{bot}}} y \;\text{ if}_{df}\; (x = \mathbf{bot}) \text{ or } (x, y \in \mathsf{CArg} \text{ and } x \sqsubseteq_{\mathsf{CArg}} y)$
$x \sqcup_{\mathbf{CArg_{bot}}} y \;=_{df}\; \textit{if } x = \mathbf{bot} \textit{ then } \mathbf{bot} \textit{ else } x \sqcup_{\mathsf{CArg}} y$
$\bot_{\mathbf{CArg_{bot}}} \;=_{df}\; \mathbf{bot}$
$\top_{\mathbf{CArg_{bot}}} \;=_{df}\; \mathsf{CArg}$

With this new domain we can construct a partial function space in which **bot** represents the undefined function value.

$\mathbf{PTrans} =_{df} \mathsf{CArg} \rightsquigarrow \mathbf{CArg_{bot}}$

As usual, Order, meet, bottom and top are defined pointwise for **PTrans** and for **PTrans**$^n$. In particular,

$\bot_{\mathbf{PTrans}^n} =_{df} \mathit{vector}\big(\lambda i{:}[1..n]\ .\ \lambda x{:}\mathsf{CArg}\ .\ \mathbf{bot}\big).$

As in [8], $\phi_i(x) = \mathbf{bot}$ indicates that the $i$'th predicate has not been called on $x$. $\phi_i(x) = \bot$ indicates that the $i$'th predicate has been called on $x$, but hasn't succeeded.

```
$\mathbf{MFG} : \texttt{Prog} \times \wp(\texttt{Literal}) \rightarrow \mathbf{PTrans}^n$
$\mathbf{MFG}[[\texttt{Pr}, \texttt{Queries}]] \ =_{df} \mathit{vector}\big(\ \lambda i{:}[1..n]\ .\ \lambda x{:}\mathsf{CArg}\ .$
    $\mathit{if}\ x \in \big(\mathbf{Reachable}[[\texttt{Pr}, \texttt{Queries}\ ]]\big)_i$
    $\mathit{then}\ \big(\mathbf{TFG}[[\texttt{Pr}]]\big)_i(x)$
    $\mathit{else}\ \mathbf{bot}\big)$
```

**Reachable** generates the domain on which **MFG**[[Pr, Queries]] is defined (not **bot**). For each of the $n$ predicates, this is a subset of CArg. Recall that, for simplicity, we assume that each query is a single literal.

```
$\mathbf{Reachable}$: $\texttt{Prog} \times \wp(\texttt{Literal}) \rightarrow (\wp(\mathsf{CArg}))^n$
$\mathbf{Reachable}[[\texttt{Pr}, \texttt{Queries}]] \ =_{df} \mathit{fix}\ \lambda \mathsf{R}{:}(\wp(\mathsf{CArg}))^n\ .$
    $\mathbf{Iterate_{Reachable}}[[\texttt{Pr}, \texttt{Queries}]](\mathsf{R})$

$\mathbf{Iterate_{Reachable}} : \texttt{Prog} \times \wp(\texttt{Literal}) \rightarrow (\wp(\mathsf{CArg}))^n \rightarrow (\wp(\mathsf{CArg}))^n$
$\mathbf{Iterate_{Reachable}}[[\texttt{Pr}, \texttt{Queries}]](\mathsf{R}) \ =_{df} \mathit{vector}\big(\lambda i{:}[1..n]\ .$
    $\big\{x{:}\mathsf{CArg} |\ \exists y \in \mathsf{R}_j,$
        $x \in \big(\mathbf{CallsMade}[[\texttt{H :- B}_1,...,\texttt{B}_m]](y,\ \mathbf{TFG}[[\texttt{Pr}]])\big)_i,$
        $\texttt{H :- B}_1,...,\texttt{B}_m$ is a clause in Pr for the $j$'th predicate. $\big\}$
    $\cup \big\{\ \big(\mathsf{GetArg}[[\texttt{Queries}]]\big)_i\ \big\}\ \big)$

$\mathbf{CallsMade}$: $\texttt{Clause} \rightarrow \mathsf{CArg} \times \mathbf{Trans}^n \rightarrow (\wp(\mathsf{Arg}))^n$
$\mathbf{CallsMade}[[\texttt{H :- B}_1,...,\texttt{B}_m]](x,\ \phi) \ =_{df} \mathit{vector}\big(\lambda i{:}[1..n]\ .$
    $\big\{y : \mathsf{CArg} |\ \mathsf{y} = \mathsf{Canon}(\mathsf{Project}(\ell,\mathbf{Trace}($
            $\mathsf{Apply}(0,\mathsf{GetEnv}[[\texttt{H :- B}_1,...,\texttt{B}_m]],x,<\texttt{H :- B}_1,...,\texttt{B}_m>),$
            $<\texttt{H :- B}_1,...,\texttt{B}_{\ell-1}>,1,\phi))),$
        $y \neq \bot,$
        $\ell \in [1..m]$, and
        $\mathit{index}(\texttt{B}_\ell) = i\ \big\}\ \big)$
```

$\mathbf{Iterate_{Reachable}}$ is continuous in R because it looks at each member of R in isolation.

## 3 The Partial Function Graph Semantics

We now present the *partial function graph* (*pfg*) semantics. It is the limit of an iteration function that is not monotonic, but that is sufficiently well behaved that the limit exists. In Section 3.2 we will show that this semantics essentially agrees with the *mfg* semantics. The reason for considering this semantics is that it is almost as informative as the *mfg* semantics and, for finite abstract domains, can be computed directly, obviating the necessity of computing the *tfg*, which for large abstract domains, would be impractical.

$(\mathbf{TFG}[[\mathtt{Pr}]])_i$ is defined on all of CArg, which may be very large. Granted, if we are to guarantee termination of our analysis, we expect to have to require that $\mathsf{CArg}_{\mathsf{approx}}$ be finite, since otherwise it is not clear how to guarantee that **Reachable**[[`Pr`, `Queries`]] is finite. For example, if we have the clause

trouble(X) :- trouble(f(X,X)).

and pose the query

?- trouble(X)

then {trouble(X)}, {trouble(f(X,X))}, {trouble(f(f(X,X),f(X,X)))}, ... would all be reachable activation sets.

Since it appears that $\mathsf{CArg}_{\mathsf{approx}}$ must be finite, we can expect **TFG**[[`Pr`]] to be computable in the approximate interpretation. But we can also expect it to be impractical for large approximation domains, which would afford greater accuracy.

### 3.1 Construction of the PFG

The *pfg* is defined as the limit of an iteration function that is almost identical to the *tfg* iteration function, except that it incorporates reachability. As the *pfg* iteration function is applied repeatedly, the domain on which the result is defined grows to reflect new reachable activations.

The *pfg* iteration function is not monotonic. But this section proves that it is sufficiently well behaved that the limit exists.

Before we can use them in the construction of $\mathbf{Iterate}_{\mathbf{PFG}}$ we must extend the types over which several functions are defined. **UseClause**, **Trace**, and **CallsMade** all have one argument of type **Trans**. Each of these arguments will now be treated as having type **PTrans**. The change in these core-semantic functions can be handled by making one small additional requirement of one of the interpretation's basic functions, FreeInst, since this is the function that may now be applied to **bot**. We require that FreeInst(**bot**) $= \perp$. With these changes in place, we can pass $\phi \in$ **PTrans** to **CallsMade** and to **UseClause** with impunity.

$$\mathbf{PFG} : \mathtt{Prog} \times \mathtt{Queries} \longrightarrow \mathbf{PTrans}^n$$

```
$\mathbf{PFG}[[\texttt{Pr}, \texttt{Queries}]] \;=_{df}\; \lim_\gamma \mathbf{Iterate_{PFG}}[[\texttt{Pr}, \texttt{Queries}]]^\gamma(\perp_{\mathbf{PTrans}^n})$
    where $f^0(X) \;=_{df}\; X$,
    $f^{\delta+1}(X) \;=_{df}\; f \circ f^\delta(X)$, and
    $f^\gamma \;=_{df}\; \bigsqcup\{f^\delta | \delta < \gamma\}$ for limit ordinals, $\gamma$.

$\mathbf{Iterate_{PFG}}$: Prog $\times$ $\wp$(Literal) $\rightarrow$ PTrans $\rightarrow$ PTrans
$\mathbf{Iterate_{PFG}}[[\texttt{Pr}, \texttt{Queries}]](\phi) \;=_{df}\; vector\big(\lambda i{:}[1..n]\ .\ \lambda x{:}\mathsf{CArg}\ .$
    if $\phi_i(x) \neq \mathbf{bot}$
    then $\bigsqcup_{\mathsf{Arg}} \{$ UseClause[[H :- $B_1,\ldots,B_m$]]$(x,\phi)$ |
        H :- $B_1,\ldots,B_m$ is a clause for the $i$'th predicate in Pr$\}$
    else if $\big((\exists y \in \mathsf{CArg})$ and $(\exists j \in [1..n])$ and
            $(\exists$ <H :- $B_1,\ldots,B_m$> $\in$ Pr$)$ such that
            $\phi_j(y) \neq \mathbf{bot}$,
            H :- $B_1,\ldots,B_m$ is a clause for the $j$'th predicate and
            $x \in (\mathbf{CallsMade}[[\texttt{H :- } B_1,\ldots,B_m]](y,\phi))_i\big)$ or
        $\big(x \in \texttt{Queries}\big)$
    then $\perp$
    else bot
```

The *then* part of $\mathbf{Iterate_{PFG}}$ is identical to $\mathbf{Iterate_{TFG}}$. On the reachable calls **PFG**[[Pr, Queries]] will give the same results as will **TFG**[[Pr]] (see Section 3.2).

The *else* part closely resembles $\mathbf{Iterate_{Reachable}}$, though $\mathbf{Iterate_{Reachable}}$ uses the fixpoint, **TFG**[[Pr]], where $\mathbf{Iterate_{PFG}}$ uses the partial solution (in a Kleene's sequence calculation), which is not complete. So, for example, in the precise interpretation, if the result instance set for $B_1$ is incomplete *vis-à-vis* the fixpoint, an incomplete activation set for $B_2$ may be reported by **CallsMade**. This is how it can happen that

$$\mathbf{MFG}[[\texttt{Pr}, \texttt{Queries}]] \overset{\sqsubseteq}{\neq}_{\mathbf{PTrans}^n} \mathbf{PFG}[[\texttt{Pr}, \texttt{Queries}]].$$

As has been mentioned already, this difference is insignificant for applications that require information about the reachability of individual activations, rather than the sets of activations that arise in our particular model.

Unfortunately, $\mathbf{Iterate_{PFG}}$ is not continuous. The sequence whose limit defines **PFG**[[Pr, Queries]] may not converge by step $\omega$. However, as long as the approximation domain is finite, the limit is certain to be reached at a finite stage.

To ensure that **PFG**[[Pr,Queries]] is well defined, we demonstrate in Lemma 1 that repeated application of $\mathbf{Iterate_{PFG}}$[[Pr,Queries]] to $\perp_{\mathbf{PTrans}^n}$ generates an increasing sequence. Then the sequence must converge before the index reaches the successor cardinal to the cardinality of **PTrans**. This proves that the limit exists and that **PFG** is well defined.

We use the notation $f|_S$ to denote restriction of the function $f$ to the set $S$. We also use this notation with equal sized vectors of functions and sets to denote the pointwise restriction.

For convenience, let us define

$$\begin{aligned} \psi_\gamma &=_{df} \mathbf{Iterate_{PFG}}[[\texttt{Pr}, \texttt{Queries}]]^\gamma(\bot_{\mathbf{PTrans}^n}), \text{ and} \\ RS_\gamma &=_{df} \mathit{vector}\Big(\lambda i \in [1..n] \ . \ \{x{:}\mathsf{CArg} | (\psi_\gamma)_i(x) \neq \mathbf{bot}\}\Big). \end{aligned}$$

At this point we begin to omit the subscripts from $\sqcup_{\mathbf{PTrans}^n}$ and $\sqsubseteq_{\mathbf{PTrans}^n}$, and the syntactic arguments from $\mathbf{Iterate_{PFG}}[[\texttt{Pr}, \texttt{Queries}]]$ and $\mathbf{Iterate_{TFG}}[[\texttt{Pr}, \texttt{Queries}]]$, etc.

**Lemma 1.** $\forall \delta < \gamma \ . \ \psi_\delta \sqsubseteq_{\mathbf{PTrans}^n} \psi_\gamma$.
We show this by induction on $\gamma$. The basis is vacuous, and the result is obvious for $\gamma$ a limit ordinal. Let us verify that it holds for successor ordinals, $\gamma + 1$. For this it suffices to show that $\psi_\gamma \sqsubseteq \psi_{\gamma+1}$.

We have, $\forall \delta < \gamma \ . \ \psi_\delta \sqsubseteq \psi_\gamma$ by the induction hypothesis. Let us assume that $\delta < \gamma$.
$\mathbf{Iterate_{TFG}}(\psi_\delta) \sqsubseteq \mathbf{Iterate_{TFG}}(\psi_\gamma)$, by monotonicity of $\mathbf{Iterate_{TFG}}$.
Since $\mathbf{Iterate_{TFG}}(\psi_\beta)|_{RS_\beta} = \mathbf{Iterate_{PFG}}(\psi_\beta)|_{RS_\beta}$,
it follows that $\mathbf{Iterate_{PFG}}(\psi_\delta)|_{RS_\delta} \sqsubseteq \mathbf{Iterate_{PFG}}(\psi_\gamma)|_{RS_\delta}$, since $RS_\delta \subseteq RS_\gamma$.
On the other hand, $x \notin (RS_\delta)_i \Rightarrow \mathbf{Iterate_{PFG}}(\psi_\delta)_i(x) \sqsubseteq \bot$,
and by inspection of the definition of $\mathbf{Iterate_{PFG}}$ we see that
$\mathbf{Iterate_{PFG}}(\psi_\delta)_i(x) = \bot \Rightarrow \bot \sqsubseteq \mathbf{Iterate_{PFG}}(\psi_\gamma)_i(x)$
So, $\mathbf{Iterate_{PFG}}(\psi_\delta)|_{\overline{RS_\delta}} \sqsubseteq \mathbf{Iterate_{PFG}}(\psi_\gamma)|_{\overline{RS_\delta}}$ follows.[7]
Putting these together, we have that $\mathbf{Iterate_{PFG}}(\psi_\delta) \sqsubseteq \mathbf{Iterate_{PFG}}(\psi_\gamma) = \psi_{\gamma+1}$.
Now, if $\gamma = \delta + 1$ for some $\delta$, we are done.
On the other hand, if $\gamma$ is a limit ordinal, then $\psi_\gamma = \sqcup_{\delta<\gamma}\mathbf{Iterate_{PFG}}(\psi_\delta)$.
But, since $\forall \delta < \gamma \ . \ \mathbf{Iterate_{PFG}}(\psi_\delta) \sqsubseteq \mathbf{Iterate_{PFG}}(\psi_\gamma) = \psi_{\gamma+1}$,
it follows that $\psi_\gamma \sqsubseteq \psi_{\gamma+1}$. ∎

## 3.2 The PFG is Safe

Now we argue that **PFG** is a safe approximation to **MFG**. It will be shown that over the calls specified by **Reachable**, all three of **MFG**, **PFG**, and **TFG** agree exactly. This is sufficient, since elsewhere **MFG**, is **bot**.

First, let us define $RS \in (\wp(\mathsf{CArg}))^n$ to represent the reachable activations according to **PFG**:

$$RS =_{df} \mathit{vector}\Big(\lambda i \in [1..n] \ . \ \{x{:}\mathsf{CArg} | \mathbf{PFG}[[\texttt{Pr}, \texttt{Queries}]]_i(x) \neq \mathbf{bot}\}\Big).$$

So, $RS = \lim_\gamma RS_\gamma$.

[7] $\overline{RS_\delta}$ denotes the pointwise compliment of $RS_\delta$.

Now we state the theorem that ensures the safeness of the *pfg* semantics as an approximation to the *mfg* semantics. We again elide the operator subscripts and syntactic arguments.

**Theorem.** $\mathbf{MFG}|_{\mathbf{Reachable}} = \mathbf{PFG}|_{\mathbf{Reachable}} = \mathbf{TFG}|_{\mathbf{Reachable}}$.
We show that $\mathbf{PFG}|_{RS} = \mathbf{TFG}|_{RS}$. Since it is obvious that $\mathbf{Reachable} \subset RS$, the Theorem will follow from this and the definition of **MFG**.

**Claim:** $\mathbf{PFG}|_{RS} \sqsubseteq \mathbf{TFG}|_{RS}$.
Since $\mathbf{PFG} =_{df} \lim_\gamma \psi_\gamma$, it suffices to show that $\psi_\gamma \sqsubseteq \mathbf{TFG}$ for all $\gamma$. This is done by transfinite induction on $\gamma$.
i. Basis: $\psi_0 \sqsubseteq \mathbf{TFG}$. Trivial.
ii. Successor Ordinals: $\psi_\delta \sqsubseteq \mathbf{TFG} \Rightarrow \psi_{\delta+1} \sqsubseteq \mathbf{TFG}$.
$\psi_{\delta+1} = \mathbf{Iterate_{PFG}}(\psi_\delta)$ by definition of $\psi_\gamma$.
$\mathbf{Iterate_{PFG}}(\psi_\delta) \sqsubseteq \mathbf{Iterate_{TFG}}(\psi_\delta)$
because they are equal on $\{x|\psi_\delta(x) \neq \mathbf{bot}\}$ by inspection,
and elsewhere it is trivially true.
$\mathbf{Iterate_{TFG}}(\psi_\delta) \sqsubseteq \mathbf{Iterate_{TFG}}(\mathbf{TFG})$
by the induction hypothesis and monotonicity of $\mathbf{Iterate_{TFG}}$.
But, $\mathbf{Iterate_{TFG}}(\mathbf{TFG}) = \mathbf{TFG}$ by definition.
iii. Limit Ordinals: $(\forall \delta < \gamma \,.\, \psi_\delta \sqsubseteq \mathbf{TFG}) \rightarrow (\psi_\gamma \sqsubseteq \mathbf{TFG})$.
This follows immediately.

**Claim:** $\mathbf{TFG}|_{RS} \sqsubseteq \mathbf{PFG}|_{RS}$
If $x \in RS_i$ then for all clauses, $\mathbf{CallsMade}[[\mathtt{H} \text{ :- } \mathtt{B}_1,\ldots,\mathtt{B}_m]](x, \mathbf{PFG})_j \subset RS_j$, for all $j \in [1..n]$.
Let $\widehat{\mathbf{PFG}}$ be **PFG** except values of **bot** are replaced by $\perp$.
Observe that $(\mathbf{Iterate_{TFG}}(\widehat{\mathbf{PFG}}))|_{RS} = \widehat{\mathbf{PFG}}|_{RS} = \mathbf{PFG}|_{RS}$.
Since **TFG** is the least fixpoint of $\mathbf{Iterate_{TFG}}$, we have the desired result.
∎

Of course, **MFG** could be constructed from **PFG** by performing reachability analysis.

This section has shown that **PFG** is a safe approximation to **MFG**. That is important because **MFG** would be rather expensive to compute for large abstract domains because it entails computing **TFG**. **PFG** is less expensive to compute because results do not have to be collected on all $x \in$ CArg, just on those that appear to be reachable.

Not only is **PFG** safe, but it is a strong approximation. Where it is defined, it agrees exactly with **TFG**. It is just defined on some $x \in$ CArg that **MFG** is not defined on. However, such $x$ always represent subsets of sets on which **MFG** *is* defined. Experiments that will be reported elsewhere seem to suggest that results are collected by the *pfg* method on relatively few $x \in$ Arg that are not reachable.

## 4 Conclusion

This paper has constructed an abstract interpretation framework for logic programs based on a *minimal function graph (mfg)* semantics. Our framework has been designed to accommodate large abstract domains efficiently, making stronger automatic inferences practical. This is the primary basis of our argument that our framework is more practical than the elegant framework of Jones and Søndergaard[7].

We have used our framework to construct and implement a data flow analysis for mode inference that gives stronger inferences than any published analysis. That result will be reported in a forthcoming paper.

Our *mfg* cannot be computed directly. But we have shown that the *partial function graph (pfg)*, which is defined here and can be computed directly for finite approximation domains, is virtually indistinguishable from the *mfg*.

## References

[1] Jung-Herng Chang, Alvin M. Despain, Doug DeGroot, "AND-Parallelism of Logic Programs Based On a Static Data Dependency Analysis," IEEE 1985 Spring CompCon.

[2] John S. Conery, "The AND/OR Process Model for Parallel Interpretation of Logic Programs," Ph.D. Dissertation, Department of Information and Computer Science, University of California-Irvine, TR 204, 1983.

[3] Patrick Cousot and Radhia Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, 1977.

[4] Saumya K. Debray, "Synthesizing Control Strategies for AND-Parallel Logic Programs," Department of Computer Science technical report, TR 87-12, University of Arizona, 1987.

[5] Saumya K. Debray, "Approximation Domains for Efficient Flow Analysis of Logic Programs," Department of Computer Science technical report, TR 87-9, University of Arizona, 1987.

[6] Saumya K. Debray, David S. Warren, "Automatic Mode Inference for Prolog Programs," *Proceedings of the 1986 Symposium on Logic Programming.*

[7] Neil D. Jones, Harald Søndergaard, "A Semantics-Based Framework for the Abstract Interpretation of Prolog," To appear in S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages,* Ellis Horwood.

[8] Neil D. Jones, Alan Mycroft, "Data Flow Analysis of Applicative Programs Using Minimal Function Graphs: Abridged Version," *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, 1986.

[9] Laxmikant Vasudeo Kalé, "Parallel Architectures for Problem Solving," Ph.D. Dissertation, University of New York at Stony Brook, 1985.

[10] J. W. Lloyd,*Foundations of Logic Programming*, Springer-Verlag, 1984.

[11] C. S. Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs," Research Report 163, Department of Artificial Intelligence, University of Edinburgh, 1981.

[12] C. S. Mellish, "Abstract Interpretation of Prolog Programs," *Third International Conference on Logic Programming*, LNCS 225, Springer-Verlag, 1986.

[13] David A. Plaisted, "The Occur-Check Problem in Prolog," *1984 International Symposium on Logic Programming,* 1984.

[14] H. Søndergaard, "An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction," *ESOP 86*, LNCS 213, Springer-Verlag, 1986.

[15] D.H.D. Warren, "Implementing Prolog - Compiling Predicate Logic Programs," Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.