Notes for a Tutorial on Abstract Interpretation of Logic Programs

Kim Marriott and Harald Søndergaard

October 1989

Overview

The present notes are concerned with semantics-based dataflow analysis of definite clause logic programs. They have been produced for a tutorial given by the authors to the North American Conference on Logic Programming in Cleveland, Ohio, 16 October 1989. The notes are a condensed version of two forthcoming papers [33, 36]. Proofs omitted here appear in these papers.

In Section 1 we give a brief introduction and historical background to the subject. In Section 2 we introduce some preliminary notation. In Section 3 we give a general theory for dataflow analysis which is basically that of abstract interpretation as introduced by P. and R. Cousot. We develop a simple abstract interpretation based on the well-known $\mathbf{T_P}$ semantics of definite clause programs. In Section 4 we consider the abstract interpretation of definite clause logic programs and detail its uses. We discuss the limitations of dataflow analyses which are based on either the $\mathbf{T_P}$ or SLD semantics of logic programs and develop a denotational semantics which may be used as a basis for most existing dataflow analyses. In Section 5 a non-trivial dataflow analysis for groundness propagation is developed from the denotational definitions given in Section 4.

1 Introduction

Dataflow analysis is an essential component of many programming tools. One use of dataflow information is to identify errors in a program, as done by program "debuggers" and type checkers. Another is in compilers and other program transformers, where the analysis may guide various optimisations.

Abstract interpretation formalizes dataflow analysis by viewing it as *approximate computation*, in which computation is performed with descriptions of data rather than the data themselves. The idea of performing program analysis by approximate computation appeared very early in computer science. Naur identified the idea and applied it in work on the Gier Algol compiler [42]. He coined the term *pseudo-evaluation* for what would later be described as "a process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not on their values" [20]. The same basic idea is found in work by Reynolds [44] and by Sintzoff [47]. Sintzoff used it for proving a number of well-formedness aspects of programs in an imperative language, and for verifying termination properties.

By the mid seventies, efficient dataflow analysis had been studied rather extensively by researchers such as Kam, Kildall, Tarjan, Ullman, and others (for references, see Hecht's book [17]). In an attempt to unify much of that work, a precise framework for discussing approximate computation (of imperative programs) was developed by Patrick and Radhia Cousot [7, 8]. The advantage of such a unifying framework is that it serves as a basis for understanding various dataflow analyses better, including their interrelation, and for discussing their correctness.

The overall idea of P. and R. Cousot was to define a "sticky" semantics which associates with each program point the set of possible storage states that may obtain at run-time whenever execution reaches that point (P. and R. Cousot called this semantics a "static" semantics). A dataflow analysis can then be construed as a finitely computable approximation to the sticky semantics. We detail this general idea in Section 3.

The work of P. and R. Cousot has later been extended to declarative languages. Such extensions are not straightforward. For example, there is no clear-cut notion of "program point" in functional or logic programs. Also, dataflow analysis of programs written in a language like Prolog differs somewhat from the analysis of programs written in more conventional languages because the dataflow is bi-directional, owing to unification, and the control flow is more complex, owing to backtracking.

Of the applications of abstract interpretation in functional programming we mention work by Jones and by Jones and Muchnick on termination analysis for lambda expressions and, in a Lisp setting, improved storage allocation schemes through reduced reference counting [21, 22]. The main application, though, has been *strictness analysis*, which is concerned with the problem of determining cases where applicative order may be safely used instead of normal order execution. The study of strictness analysis was initiated by Mycroft [40] and the literature on the subject is now quite extensive, see for example Nielson [43].

Abstract interpretation of logic programs was first mentioned by Mellish [37] where it was suggested as a way to formalize mode analysis. This suggestion can be attributed to Alan Mycroft. Søndergaard [48] gave an occur check analysis which was formalized as an abstract interpretation. A general framework for the abstract interpretation of logic programs was first given by Mellish [38]. Debray further investigated abstract interpretation for mode analysis [12]. A general framework for the abstract interpretation of logic programs based on a denotational definition of SLD was given by Jones and Søndergaard [25], a framework based on AND-OR trees was given by Bruynooghe *et al.* [6] and one based on OLDT resolution was given by Kanamori and Kawamura [26]. A framework for the abstract interpretation of normal programs based on the three-valued logic of Kleene [27] was given by Marriott and Søndergaard [31, 32]. A unified treatment of various frameworks is given by Marriott and Søndergaard [35].

For a general introduction to abstract interpretation and an extensive list of references, we refer to the book edited by Abramsky and Hankin [1].

2 General Preliminaries

In this section we recapitulate some basic notions and facts from domain theory and explain some notation that will be used throughout. For a detailed introduction the reader is referred to a textbook, for example Schmidt's [46] or Birkhoff's book on lattice theory [4].

Let $\mathbf{I}$ denote the identity relation on a set $\mathbf{X}$. A *preordering* on $\mathbf{X}$ is a binary relation $\mathbf{R}$ that is reflexive ($\mathbf{I} \subseteq \mathbf{R}$) and transitive ($\mathbf{R} \cdot \mathbf{R} \subseteq \mathbf{R}$). A *partial ordering* is a preordering that is antisymmetric ($\mathbf{R} \cap \mathbf{R}^{-1} \subseteq \mathbf{I}$). A set equipped with a partial ordering is a *poset.* Let $(\mathbf{X}, \leq)$ be a poset. A (possibly empty) subset $\mathbf{Y}$ of $\mathbf{X}$ is a *chain* iff for all $\mathbf{y}, \mathbf{y}' \in \mathbf{Y}, \mathbf{y} \leq \mathbf{y}' \vee \mathbf{y}' \leq \mathbf{y}$. We let $\odot\,\mathbf{X}$ denote the poset $\mathbf{X}$ extended with an element $\perp$ which is least, that is $\perp \leq \mathbf{x}$ for all $\mathbf{x} \in \odot\,\mathbf{X}$.

Let $(\mathbf{X}, \leq)$ be a poset. An element $\mathbf{x} \in \mathbf{X}$ is an *upper bound* for $\mathbf{Y}$ iff $\mathbf{y} \leq \mathbf{x}$ for all $\mathbf{y} \in \mathbf{Y}$. Dually we may define a *lower bound* for $\mathbf{Y}$. An upper bound $\mathbf{x}$ for $\mathbf{Y}$ is the *least upper bound* for $\mathbf{Y}$ iff, for every upper bound $\mathbf{x}'$ for $\mathbf{Y}$, $\mathbf{x} \leq \mathbf{x}'$, and when it exists, we denote it by $\bigsqcup \mathbf{Y}$. Dually we may define the *greatest lower bound* $\bigsqcap \mathbf{Y}$ for $\mathbf{Y}$.

A poset for which all subsets possess least upper bounds and greatest lower bounds is a *complete lattice.* In particular, equipped with the subset ordering, the powerset of $\mathbf{X}$, denoted by $\mathcal{P}\,\mathbf{X}$, is a complete lattice. Let $\mathbf{X}$ be a complete lattice. We denote $\bigsqcup = \bigsqcap \mathbf{X}$ by $\perp_{\mathbf{X}}$ and $\bigsqcap = \bigsqcup \mathbf{X}$ by $\top_{\mathbf{X}}$. In dataflow analysis we are often interested in complete lattices which are "ascending chain finite." The complete lattice $\mathbf{X}$ is *ascending chain finite* (or *Noetherian*) iff every non-empty subset $\mathbf{Y} \subseteq \mathbf{X}$ has an element that is maximal in $\mathbf{Y}$, or equivalently, for every non-empty chain $\mathbf{Y} \subseteq \mathbf{X}$, $\bigsqcup \mathbf{Y} \in \mathbf{Y}$.

We write $\bigsqcup_{(\mathbf{Q}\ \mathbf{x})} \mathbf{F}\ \mathbf{x}$ for $\bigsqcup\{\mathbf{F}\ \mathbf{x} \mid \mathbf{Q}\ \mathbf{x}\}$, where $\mathbf{Y}$ is a complete lattice, $\mathbf{F} : \mathbf{X} \to \mathbf{Y}$, and $\mathbf{Q}$ is some predicate on $\mathbf{X}$. In particular, we use $\bigcup_{(\mathbf{Q}\ \mathbf{x})} \mathbf{F}\ \mathbf{x}$ for $\bigcup\{\mathbf{F}\ \mathbf{x} \mid \mathbf{Q}\ \mathbf{x}\}$, where $\mathbf{F} : \mathbf{X} \to \mathcal{P}\,\mathbf{Y}$ and $\mathbf{x} \in \mathbf{X}$ for some $\mathbf{X}$ and $\mathbf{Y}$.

Functions are generally used in their Curried form. Our notation for function application uses parentheses sparingly. Only when it would seem to help the eye, shall we make use of redundant parentheses. As usual, function space formation $\mathbf{X} \to \mathbf{Y}$ associates to the right, and function application to the left.

Let $\mathbf{F} : \mathbf{X} \to \mathbf{Y}$ be a function. Then $\mathbf{F}$ is *injective* iff $\mathbf{F}\ \mathbf{x} = \mathbf{F}\ \mathbf{x}' \Rightarrow \mathbf{x} = \mathbf{x}'$ for all $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$, and $\mathbf{F}$ is *bijective* iff there is a function $\mathbf{F}' : \mathbf{Z} \to \mathbf{X}$ such that $\mathbf{F} \circ \mathbf{F}'$ and $\mathbf{F}' \circ \mathbf{F}$ are identity functions. We define $\mathbf{F}$'s distributed version to be the function $(\Delta\,\mathbf{F}) : \mathcal{P}\,\mathbf{X} \to \mathcal{P}\,\mathbf{Y}$, defined by $\Delta\,\mathbf{F}\ \mathbf{Z} = \{\mathbf{F}\ \mathbf{z} \mid \mathbf{z} \in \mathbf{Z}\}$.

For any set $\mathbf{X}$ let $\mathbf{X}*$ denote the set of finite sequences of elements of $\mathbf{X}$. The empty sequence is denoted by **nil** and we use the operator ":" for sequence construction. The *sequentialized* version of a function $\mathbf{F} : \mathbf{X} \to \mathbf{Y} \to \mathbf{Y}$ is given by application of $\Sigma : (\mathbf{X} \to \mathbf{Y} \to \mathbf{Y}) \to \mathbf{X}* \to \mathbf{Y} \to \mathbf{Y}$, defined by

$$\Sigma\ \mathbf{F\ nil\ z} = \mathbf{z}$$
$$\Sigma\ \mathbf{F\ (x : y)\ z} = \Sigma\ \mathbf{F\ y\ (F\ x\ z)}.$$

Let $(\mathbf{X}, \leq)$ and $(\mathbf{Z}, \preceq)$ be posets and let $\mathbf{F} : \mathbf{X} \to \mathbf{Z}$ be a function. The function $\mathbf{F} : \mathbf{X} \to \mathbf{Z}$ is *monotonic* iff $\mathbf{x} \leq \mathbf{x}' \Rightarrow \mathbf{F\ x} \preceq \mathbf{F\ x}'$ for all $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$. In what follows, monotonicity of functions is essential, so much so that it is understood throughout these notes that $\mathbf{X} \to \mathbf{Y}$ denotes the space of monotonic functions.

A *fixpoint* for a function $\mathbf{F} : \mathbf{X} \to \mathbf{X}$ is an element $\mathbf{x} \in \mathbf{X}$ such that $\mathbf{x} = \mathbf{F\ x}$. If $\mathbf{X}$ is a complete lattice, then the set of fixpoints for (the monotonic) $\mathbf{F} : \mathbf{X} \to \mathbf{X}$ is itself a complete lattice. The least element of this lattice is the *least fixpoint* for $\mathbf{F}$, denoted by $\mathbf{lfp\ F}$. Furthermore, defining

$$\begin{array}{lcll} \mathbf{F} \uparrow \alpha & = & \mathbf{F}\ (\mathbf{F} \uparrow (\alpha - 1)) & \text{if } \alpha \text{ is a successor ordinal} \\ \mathbf{F} \uparrow \alpha & = & \bigsqcup\{\mathbf{F} \uparrow \alpha' \mid \alpha' < \alpha\} & \text{if /}\mathbf{alpha} \text{ is a limit ordinal,} \end{array}$$

there is some ordinal α such that $\mathbf{F} \uparrow \alpha = \mathbf{lfp\ F}$. The sequence $\mathbf{F} \uparrow 0, \mathbf{F} \uparrow 1, \ldots, \mathbf{lfp\ F}$ is the *Kleene sequence* for $\mathbf{F}$.

Let $\mathbf{X}$ be a complete lattice. A predicate $\mathbf{Q}$ is *inclusive* on $\mathbf{X}$ iff for all (possibly empty) chains $\mathbf{Y} \subseteq \mathbf{X}, \mathbf{Q}\ (\bigsqcup \mathbf{Y})$ holds whenever $\mathbf{Q\ y}$ holds for every $\mathbf{y} \in \mathbf{Y}$. Inclusive predicates are admissible in *fixpoint induction.* Assume that $\mathbf{F} : \mathbf{X} \to \mathbf{X}$ is monotonic and $(\mathbf{Q\ x}) \Rightarrow \mathbf{Q}\ (\mathbf{F\ x})$ for all $\mathbf{x} \in \mathbf{X}$. If $\mathbf{Q}$ is inclusive then $\mathbf{Q}\ (\mathbf{lfp\ F})$ holds.

In these notes we will only be concerned with definite programs [28]. A *definite program*, or *program*, is a finite set of clauses. A *clause* is of the form $\mathbf{H} \leftarrow \mathbf{B}$ where $\mathbf{H}$, the *head*, is an *atom* and $\mathbf{B}$, the *body*, is a finite sequence of atoms. We let **Var** denote the (countably infinite) set of variables, **Term** the set of terms, **Pred** the set of predicate symbols, **Atom** the set of atoms, **Clause** the set of clauses, and **Prog** the set of (definite) programs.

3 Abstract Interpretation

Abstract interpretation formalizes the idea of approximate computation in which computation is performed with descriptions of data rather than the data itself. Approximate computation is well-known from everyday use. Examples are the casting out of nines to check numerical computations and application of the rules of signs, such as "plus times minus yields minus." The disadvantage of an approximate computation is that the result it yields is not, in general, as precise as those of the proper computation. However this is compensated for by the fact that the approximate computation is (usually) much faster than the proper computation. Our concern is with the approximate computation of programs. In this case, the difference in speed between proper and approximate computation may be extreme: non-termination versus termination.

We describe approximate computation more formally as evaluating a formula or a program, not over its standard domain **E**, but by using a set **D** of descriptions of objects in **E**. The domain **E** may consist of sets of terms, atomic formulas, substitutions, or whatever, depending on how the semantics is modelled, and **D** is determined by the sort of program properties we want to expose. Of course, when performing approximate computations, one must reinterpret all operators so as to apply to descriptions rather than to proper values.

Assume we have a standard domain **E** and a set **D** of descriptions. To be precise about the relation between values and descriptions, a *concretization function* $\gamma : \mathbf{D} \to \mathbf{E}$ is required. For every description $\mathbf{d} \in \mathbf{D}$, $(\gamma\ \mathbf{d})$ is the "largest" object which **d** describes. Thus γ is, in a sense, the semantic function for descriptions. We follow P. and R. Cousot by requiring the existence of an *abstraction function* $\alpha : \mathbf{E} \to \mathbf{D}$. For every object $\mathbf{e} \in \mathbf{E}$, $(\alpha\ \mathbf{e})$ is the "best," that is least, description of **e**. We require that α and γ form a "Galois insertion."

Definition. Let **D** and **E** be complete lattices and $\gamma : \mathbf{D} \to \mathbf{E}$ and $\alpha : \mathbf{E} \to \mathbf{D}$ be (monotonic) functions. Then $(\mathbf{D}, \gamma, \mathbf{E}, \alpha)$ is a *Galois insertion* iff

$$\forall \mathbf{d} \in \mathbf{D} \,.\, \mathbf{d} = \alpha\ (\gamma\ \mathbf{d}),$$
$$\forall \mathbf{e} \in \mathbf{E} \,.\, \mathbf{e} \leq \gamma\ (\alpha\ \mathbf{e}),$$

where $\leq$ is the ordering on E. ■

Descriptions are ordered according to how large a set of objects they apply to: the more imprecise, the "higher" they sit in the ordering. Note that, if they exist, α and γ uniquely determine each other.

The requirement that there exist both an abstraction and a concretization function is rather strong. In fact, one can generalize this discussion by dropping the requirement that there is a concretization function [41, 43], or dropping the requirement that there is an abstraction function [6, 32, 35]. Marriott [30] has a more detailed discussion of this.

Example 3.1 (Reachability) Assume $\mathbf{E} = \mathcal{P}\,\mathbf{U}$. Let $\mathbf{D} = \{\bot, \top\}$, $\gamma\ \bot = \emptyset$, and $\gamma\ \top = \mathbf{U}$. This simple domain can be used in a so-called reachability analysis: unreachable program points are described by $\bot$, and possibly reachable points by $\top$. ■

Example 3.2 (Standard semantics) Here $\mathbf{D} = \mathbf{E}$, the most fine-grained set of descriptions possible. The concretization and abstraction functions are both the identity function. ■

Clearly, the finer-grained descriptions we use, the better dataflow analysis possible. On the other hand, considerations of finite computability and efficiency put a limit on granularity.

In accordance with the observation that "larger" descriptions naturally correspond to decreased precision, we now define what it means for $\mathbf{d} \in \mathbf{D}$ to safely approximate $\mathbf{e} \in \mathbf{E}$.

Definition. Let $(\mathbf{D}, \gamma, \mathbf{E}, \alpha)$ be a Galois insertion. We define $\mathbf{appr}_\gamma : \mathbf{D} \times \mathbf{E} \rightarrow \mathbf{Bool}$ by

$$\mathbf{appr}_\gamma \ (\mathbf{d}, \mathbf{e}) \text{ iff } \mathbf{e} \leq \gamma \ \mathbf{d},$$

where $\leq$ is the ordering on $\mathbf{E}$. ■

Thus $\mathbf{appr}_\gamma \ (\mathbf{d}, \mathbf{e})$ reads "$\mathbf{d}$ approximates $\mathbf{e}$ under γ." Since γ will always be clear from the context, we shall omit the subscript and simply denote the predicate by **appr**. Equivalently, we can define **appr** in terms of α; $\mathbf{appr} \ (\mathbf{d}, \mathbf{e})$ iff $\alpha \ \mathbf{e} \leq \mathbf{d}$. We shall often write **appr** as an infix operator. There is a natural extension of **appr** to cartesian products and function spaces:

Definition. We extend **appr** from the domains $\mathbf{D} \times \mathbf{E}$ and $\mathbf{D}' \times \mathbf{E}'$ to

(a) $(\mathbf{D} \times \mathbf{D}') \times (\mathbf{E} \times \mathbf{E}')$ by defining: $(\mathbf{d}, \mathbf{d}') \ \mathbf{appr} \ (\mathbf{e}, \mathbf{e}')$ iff $\mathbf{d} \ \mathbf{appr} \ \mathbf{e} \wedge \mathbf{d}' \ \mathbf{appr} \ \mathbf{e}'$,

(b) $(\mathbf{D} \rightarrow \mathbf{D}') \times (\mathbf{E} \rightarrow \mathbf{E}')$ by defining:
$\mathbf{F} \ \mathbf{appr} \ \mathbf{F}'$ iff $\forall \ (\mathbf{d}, \mathbf{e}) \in \mathbf{D} \times \mathbf{E}' \, . \, \mathbf{d} \ \mathbf{appr} \ \mathbf{e} \Rightarrow (\mathbf{F} \ \mathbf{d}) \ \mathbf{appr} \ (\mathbf{F}' \ \mathbf{e})$. ■

We thus in fact have a series of relations "**appr**," but in what follows, the "type" of **appr** should always be clear from the context. This treatment of **appr** is similar to Reynold's use of relational functors [45]. For semantic functions we shall sometimes use the symbol $\propto$ to denote the approximation relation.

Definition. Let $\mathbf{F} : \mathbf{Prog} \rightarrow \mathbf{D}$ and $\mathbf{F}' : \mathbf{Prog} \rightarrow \mathbf{E}$ be semantic functions, and let $(\mathbf{D}, \gamma, \mathbf{E}, \alpha)$ be a Galois insertion. Then $\mathbf{F} \propto \mathbf{F}'$ iff $(\mathbf{F} \ \mathbf{P}) \ \mathbf{appr} \ (\mathbf{F}' \ \mathbf{P})$ holds for all $\mathbf{P} \in \mathbf{Prog}$. ■

To illustrate these ideas we give a very simple dataflow analysis for "type inference" based on the well-known $\mathbf{T_P}$ operator [2]. In this setting we regard type inference as finding an approximation to the least Herbrand model of the program. The least Herbrand model is just the least fixpoint of $\mathbf{T_P}$ so we are interested in approximating $\mathbf{lfp} \ \mathbf{T_P}$. We first recapitulate the definition of $\mathbf{T_P}$.

Let **Her** denote the set of ground atoms (for some fixed alphabet). For a syntactic object $\mathbf{s}$, $(\mathbf{ground} \ \mathbf{s})$ denotes the set of ground instances of $\mathbf{s}$. The domain of $\mathbf{T_P}$ is the set of interpretations, $\mathbf{Int} = \mathcal{P} \, \mathbf{Her}$.

Definition. The *immediate consequence function* $\mathbf{T_P} : \mathbf{Int} \to \mathbf{Int}$ is defined by

$$\mathbf{T_P\ u} = \{\mathbf{A} \in \mathbf{Her} \mid \exists\, \mathbf{C} \in \mathbf{P} \,.\, \exists [\![\mathbf{A} \leftarrow \mathbf{B}]\!] \in \mathbf{ground\ C} \,.\, \mathbf{u\ makes\ B\ true}\}.$$

where, if $\mathbf{B} = \mathbf{A}_1 : \ldots : \mathbf{A_n} : \mathbf{nil}$ is a ground body, we say that

$$\mathbf{u\ makes\ B\ true} \text{ iff } \forall\, \mathbf{i} \in \{1, \ldots, \mathbf{n}\} \,.\, \mathbf{A_i} \in \mathbf{u}.$$

The *least model semantics* of program $\mathbf{P}$ is $\mathbf{L\ P} = \mathbf{lfp\ T_P}$. ■

The descriptions in our analysis are extremely simple—just sets of predicate symbols. A predicate symbol is in a description if some ground atom with that predicate symbol is in the interpretation. For instance, the best description of $\{\mathbf{p(a)}, \mathbf{p(b)}, \mathbf{r(a)}\}$ is $\{\mathbf{p}, \mathbf{r}\}$.

Definition. Let $\mathbf{D} = \mathcal{P}\,\mathbf{Pred}$ and let $\mathbf{pred\ A}$ denote the predicate symbol of atom $\mathbf{A}$. Define $\gamma_\mathbf{D} : \mathbf{D} \to \mathbf{Int}$ by

$$\gamma_\mathbf{D}\ \mathbf{d} = \{\mathbf{A} \in \mathbf{Her} \mid \mathbf{pred\ A} \in \mathbf{d}\}$$

and $\alpha_\mathbf{D} : \mathbf{Int} \to \mathbf{D}$ by

$$\alpha_\mathbf{D}\ \mathbf{u} = \{\mathbf{Q} \in \mathbf{Pred} \mid \exists\, \mathbf{A} \in \mathbf{u} \,.\, \mathbf{pred\ A} = \mathbf{Q}\}. \quad ■$$

Proposition 3.3 *The tuple* $(\mathbf{D}, \gamma_\mathbf{D}, \mathbf{Int}, \alpha_\mathbf{D})$ *is a Galois insertion.* ■

We can approximate $\mathbf{T_P}$ using the following operator.

Definition. The operator $\mathbf{U_P} : \mathbf{D} \to \mathbf{D}$ is defined by

$$\mathbf{U_P\ d} = \{\mathbf{Q} \in \mathbf{Pred} \mid \exists [\![\mathbf{A} \leftarrow \mathbf{B}]\!] \in \mathbf{P} \,.\, \mathbf{pred\ A} = \mathbf{Q} \wedge \mathbf{d\ makes\ B\ true}\}$$

where $\mathbf{d\ makes\ A}_1 : \ldots : \mathbf{A_n} : \mathbf{nil\ true}$ iff $\forall\, \mathbf{i} \in \{1, \ldots, \mathbf{n}\} \,.\, \mathbf{pred\ A_i} \in \mathbf{d}$. The *type semantics* of program $\mathbf{P}$ is $\mathbf{T\ P} = \mathbf{lfp\ U_P}$. ■

It is straightforward to show that $\mathbf{U_P\ appr\ T_P}$ holds. Usually we want a dataflow analysis to be as precise as possible, in the sense of making the best possible use of available information. Letting an approximating function $\mathbf{F}'$ map every element of $\mathbf{D}$ to $\top_\mathbf{D}$ clearly leads to a dataflow analysis that is correct, but useless. We note that in the present framework, a *best* safe approximating function $\mathbf{F}'$ always exists, namely the function defined by $\mathbf{F}' = \alpha \circ \mathbf{F} \circ \gamma$. This is not always true if we drop the requirement that an abstraction function exists. We can show that for the descriptions $\mathbf{D}$, $\mathbf{U_P}$ is the best approximation to $\mathbf{T_P}$.

Proposition 3.4 $\mathbf{U_P} = \alpha_{\mathbf{D}} \circ \mathbf{T_P} \circ \gamma_{\mathbf{D}}$. ■

What we must now show is that $\mathbf{T} \propto \mathbf{L}$. This is true if the relation **appr** is "preserved" by the least fixpoint operator, that is, **appr** is inclusive. We note that not all seemingly reasonable predicates are inclusive, for example the predicate "is finite" is not inclusive on an infinite powerset. However in our case **appr** is inclusive.

Lemma 3.5 *The predicate* **appr** *is inclusive on* $\mathbf{D} \times \mathbf{E}$*, ordered componentwise.* ■

The following proposition is easily proved by fixpoint induction.

Proposition 3.6 *Let* $(\mathbf{D}, \gamma, \mathbf{E}, \alpha)$ *be a Galois insertion and let (monotonic)* $\mathbf{F} : \mathbf{E} \to \mathbf{E}$ *and* $\mathbf{F}' : \mathbf{D} \to \mathbf{D}$ *be such that* $\mathbf{F}'$ **appr** $\mathbf{F}$ *holds. Then* $(\mathbf{lfp}\ \mathbf{F}')$ **appr** $(\mathbf{lfp}\ \mathbf{F})$ *holds.* ■

We thus have the following proposition.

Proposition 3.7 $\mathbf{T} \propto \mathbf{L}$. ■

The reason for our interest in fixpoints is the fact that semantics for logic programs are naturally expressed as fixpoint characterizations. This applies to the $\mathbf{T_P}$ characterization, certain formulations of SLD resolution, and to denotational definitions. More precisely, the idea is to have the "standard" semantics of program **P** given as **lfp F** for some function **F**, and to have dataflow analyses defined in terms of "non-standard" functions $\mathbf{F}'$, approximating **F**. We can then use Proposition 3.6 to conclude that all elements of **lfp F** have some property **R**, provided all elements of γ $(\mathbf{lfp}\ \mathbf{F}')$ have property **R**. In other words, $\mathbf{lfp}\ \mathbf{F}'$ provides us with approximate information about the standard semantics **lfp F**. In this way dataflow analyses are nothing but approximations to the standard semantics.

Since it is preferable that approximations are finitely computable, the approximating function $\mathbf{F}'$ and the description domain are usually chosen such that the Kleene sequence for $\mathbf{F}'$ is finite. It is common to use description domains which are ascending chain finite lattices as this ensures that the Kleene sequence for $\mathbf{F}'$ is finite. While ascending chain finiteness is a sufficient condition for termination, it is not necessary. For a discussion of termination, see P. and R. Cousot [7].

Most of the above discussion has been too simplistic in one respect. For dataflow analysis purposes we are often interested in a semantics that is somewhat more complex than the standard "base" semantics which simply specifies a program's "input-output" relation. The reason is that we are looking for invariants (such as "**x** is always ground") that hold at *program points*, not just *results* of computations. A *sticky semantics* is obtained by extending the base semantics so that for each program, all run-time states associated with each of its program points are recorded. In this connection, the *base semantics* may be viewed as a degenerate sticky semantics that has only one program point, namely the "end" of the program. A dataflow analysis should be proven correct with respect to a sticky semantics rather than the base semantics.

In the case of $\mathbf{T_P}$ the sticky semantics could keep information about the ground clause instances whose bodies succeed. This information is kept in a *repository.* A repository is similar to what is sometimes called a "context vector" [7], a "log" [25, 48], or a "record" [31]. Let **Gcla** denote the set of ground clauses (for some fixed alphabet), and let the domain of repositories be $\mathbf{Rep} = \mathbf{Clause} \rightarrow \mathcal{P}\,\mathbf{Gcla}$. Ordered pointwise, **Rep** forms a complete lattice. The sticky semantics can now be defined as a fixpoint of a certain operator on the domain $\mathbf{Env} = \mathbf{Int} \times \mathbf{Rep}$. Ordered componentwise, **Env** forms a complete lattice.

Definition. Let $\mathbf{V_P} : \mathbf{Env} \rightarrow \mathbf{Env}$ be defined by

$$\mathbf{V_P}\ (\mathbf{u}, \mathbf{r}) = (\mathbf{T_P}\ \mathbf{u}, \mathbf{r} \sqcup \mathbf{r'}) \text{ where } \mathbf{r'}\ \mathbf{C} = \{[\![\mathbf{A} \leftarrow \mathbf{B}]\!] \in \mathbf{ground}\ \mathbf{C} \mid \mathbf{u}\ \mathbf{makes}\ \mathbf{B}\ \mathbf{true} \wedge \mathbf{C} \in \mathbf{P}\}.$$

The *sticky semantics* of program **P** is $\mathbf{lfp}\ \mathbf{V_P}$. ■

It is not hard to see that $\mathbf{V_P}$ is monotonic, so the sticky semantics is well-defined. Furthermore the relation to the least model semantics should be clear: $\mathbf{lfp}\ \mathbf{V_P}$ has the form $(\mathbf{u}, \mathbf{r})$ where $\mathbf{u} = \mathbf{lfp}\ \mathbf{T_P}$ is the least model semantics. We further note that $\mathbf{lfp}\ \mathbf{V_P} = \mathbf{V_P}\ (\mathbf{lfp}\ \mathbf{T_P}, \bot_{\mathbf{Rep}})$. It is straightforward to extend the type semantics to a sticky type semantics such that the sticky type semantics approximates the sticky semantics.

4 Dataflow Analysis of Logic Programs

In this section we show how abstract interpretation can be used to develop dataflow analyses for logic programs. The analysis sketched in Section 3 was based on the $\mathbf{T_P}$ semantics. We saw how it could be used for a kind of type inference, and we have shown elsewhere how this in turn may be useful for program specialization and error detection [31, 32]. One limitation of this semantics is that it does not give any information about how *variables* would actually get bound during execution of a program. Most "compiler optimizations," however, depend on knowledge about variable bindings at various stages of execution, so to support such transformations, we have to develop semantic definitions that manipulate *substitutions.* Furthermore, a typical Prolog compiler, say, is based on an execution mechanism given by SLD resolution, but the $\mathbf{T_P}$ model lacks the SLD model's notion of "calls" to clauses.

We therefore need to base our definitions on a formalization of SLD resolution. Such a formalization is given in Section 4.1. The SLD semantics, however, is not very useful as a basis for dataflow analysis, and we explain why. We therefore develop denotational semantics that are better suited. Section 4.2 introduces "parametric substitutions" as the natural semantic domain for such definitions. The definition of a "base" semantics is given in Section 4.3, and in Section 4.4 we develop, by minor changes, the semantic definition into a generic definition of a wide range of dataflow analyses. The definition is generic in the sense that it remains uncommitted as to what "substitution approximations" are and how to "compose" and "unify" them. Finally, Section 4.5 lists a series of applications of the presented (or a very similar) framework.

4.1 SLD-Based Analysis

The definition in Section 3 is useful as a basis for dataflow analyses that yield approximations to a program's success set. It is, however, not very useful as a basis for a number of the dataflow analyses that designers of compilers often are interested in. Interpreters and compilers for logic programming languages are usually based on SLD resolution as execution mechanism. The previous definition does not capture SLD resolution, because it has no notion of "calls" to clauses, as has the SLD model. In the SLD model, the first thing that takes place when a clause is called is unification of the calling atom and the clause's head. This unification is an important target for a compiler's attempts to generate more efficient code, because the general unification algorithm is expensive, and most calls only need very specialized versions of the algorithm. (There are other transformations than unification specialization that we are interested in, but this serves as sufficient motivation for incorporating "call unification" in the semantic definition).

We now formulate SLD resolution. We assume that we are given a function $\mathbf{vars} : (\mathbf{Prog} \cup \mathbf{Atom} \cup \mathbf{Term}) \to \mathcal{P}\,\mathbf{Var}$, such that $(\mathbf{vars}\ \mathbf{s})$ is the set of variables in the syntactic object $\mathbf{s}$.

A program denotes a computation in a domain of *substitutions*. A substitution is an almost-identity mapping $\theta \in \mathbf{Sub} \subseteq \mathbf{Var} \to \mathbf{Term}$ from the set of variables $\mathbf{Var}$ to the set of terms over $\mathbf{Var}$. Substitutions are not distinguished from their natural extensions to $\mathbf{Atom} \to \mathbf{Atom}$. Our notation for substitutions is standard. For instance $\{\mathbf{x} \mapsto \mathbf{a}\}$ denotes the substitution θ such that $(\theta\ \mathbf{x}) = \mathbf{a}$ and $(\theta\ \mathbf{V}) = \mathbf{V}$ for all $\mathbf{V} \neq \mathbf{x}$. We let ι denote the identity substitution. The functions $\mathbf{dom}, \mathbf{rng}, \mathbf{vars} : \mathbf{Sub} \to \mathcal{P}\,\mathbf{Var}$ are defined by

$$\begin{aligned}
\mathbf{dom}\,\theta &= \{\mathbf{V} \mid \theta\,\mathbf{V} \neq \mathbf{V}\} \\
\mathbf{rng}\,\theta &= \textstyle\bigcup_{(\mathbf{V} \in \mathbf{dom}\,\theta)} \mathbf{vars}\ (\theta\ \mathbf{V}) \\
\mathbf{vars}\,\theta &= (\mathbf{dom}\,\theta) \cup (\mathbf{rng}\,\theta).
\end{aligned}$$

A *unifier* of $\mathbf{A}, \mathbf{H} \in \mathbf{Atom}$ is a substitution θ such that $(\theta\ \mathbf{A}) = (\theta\ \mathbf{H})$. A unifier θ of $\mathbf{A}$ and $\mathbf{H}$

is an (idempotent) *most general unifier* of $\mathbf{A}$ and $\mathbf{H}$ iff $\theta' = \theta' \circ \theta$ for every unifier θ' of $\mathbf{A}$ and $\mathbf{H}$. The auxiliary function $\mathbf{mgu} : \mathbf{Atom} \to \mathbf{Atom} \to \mathcal{P}\,\mathbf{Sub}$ is defined as follows. If $\mathbf{A}$ and $\mathbf{H}$ are unifiable, then ($\mathbf{mgu\ A\ H}$) yields a singleton set consisting of a most general unifier of $\mathbf{A}$ and $\mathbf{H}$. Otherwise ($\mathbf{mgu\ A\ H}$) $= \emptyset$.

The function $\mathbf{restrict} : \mathcal{P}\,\mathbf{Var} \to \mathbf{Sub} \to \mathbf{Sub}$ is defined by

$$\mathbf{restrict\ U}\ \theta\ \mathbf{V} \quad = \quad \text{if } \mathbf{V} \in \mathbf{U} \text{ then } \theta\ \mathbf{V} \text{ else } \mathbf{V}.$$

We also use a function $\mathbf{rename} : \mathcal{P}\,\mathbf{Var} \to \mathbf{Var} \to \mathbf{Var}$ for renaming: ($\mathbf{rename\ U}$) is some bijective substitution, or *renaming*, such that ($\mathbf{rename\ U\ V}$) $\notin \mathbf{U}$ for all $\mathbf{V} \in \mathbf{Var}$. We shall use the extension of ($\mathbf{rename\ U}$) to $\mathbf{Clause} \to \mathbf{Clause}$.

We will assume the standard (left-to-right) computation rule. As we discuss shortly, our dataflow analyses will not distinguish non-termination from finite failure. As a result of this, we can assume a *parallel* search rule, rather than the customary depth-first rule—this simplifies our task. Owing to the use of a parallel search rule, the execution of a program naturally yields a *set* of answer substitutions.

We now give a definition of the SLD semantics of a definite program. The definition is given in a form that should make it clear that it is equivalent to the usual SLD model [2, 28]. We let :: denote concatenation of sequences. Clauses are implicitly universally quantified, so each call of a clause should produce new incarnations of its variables. This is done by means of the function **rename**, so that the generated clause instance will contain no variables previously met during execution (those in θ) or to be met later (those in $\mathbf{A} :: \mathbf{G}$). As usual, we think of a program as a set of clauses, and of goals and bodies as sequences of atoms.

Definition. The *SLD semantics* has semantic function $\mathbf{O} : \mathbf{Prog} \to \mathbf{Atom}{*} \to \mathcal{P}\,\mathbf{Sub}$ and is defined as follows.

$$\begin{array}{lcl}
\mathbf{O\ P\ G} & = & \Delta\ (\mathbf{restrict}\ (\mathbf{vars\ G}))\ (\mathbf{F_P\ G}\ \iota) \\
\mathbf{F_P\ nil}\ \theta & = & \{\theta\} \\
\mathbf{F_P}\ (\mathbf{A} : \mathbf{G})\ \theta & = & \bigcup_{\mathbf{C} \in \mathbf{P}} \text{let } [\![\mathbf{H} \leftarrow \mathbf{B}]\!] = \mathbf{rename}\ ((\mathbf{vars}\ \theta) \cup (\mathbf{vars}\ (\mathbf{A} : \mathbf{G})))\ \mathbf{C} \text{ in} \\
 & & \qquad \text{let } \Theta = \mathbf{mgu\ A\ H} \text{ in } \bigcup_{\theta' \in \Theta} \mathbf{F_P}\ (\theta'\ (\mathbf{B} :: \mathbf{G}))\ (\theta' \circ \theta). \quad \blacksquare
\end{array}$$

Note that finite failure is not distinguished from non-termination. For example, let $\mathbf{P}$ be the program consisting only of the clause $\mathbf{p}$. Let $\mathbf{P'}$ be the program consisting only of the clause $\mathbf{q} \leftarrow \mathbf{q}$, and consider the query $\leftarrow \mathbf{q}$. We have that $\mathbf{O\ P\ q} = \mathbf{O\ P'\ q} = \emptyset$. This is not a flaw in the semantic definition, it merely reflects the fact that we are not interested in the distinction, since termination issues are often disregarded in dataflow analysis. The statements we generate are of the form "whenever execution reaches this point, so and so holds." But in saying so, we do not actually say that execution does reach the point. In particular, "whenever the computation terminates, so and so holds" concludes nothing about termination.

Example 4.1 Consider the following list concatenation program $\mathbf{P}$:

$$\begin{array}{l} \mathbf{append}(\mathbf{nil}, \mathbf{y}, \mathbf{y}). \\ \mathbf{append}(\mathbf{u} : \mathbf{x}, \mathbf{y}, \mathbf{u} : \mathbf{z}) \leftarrow \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z}). \end{array}$$

and the query $\mathbf{G} = \leftarrow\mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{a} : \mathbf{nil})$. Execution of $\mathbf{P}$ yields two instantiations of the variables in $\mathbf{G}$. We have that $\mathbf{O}\ \mathbf{P}\ \mathbf{G} = \{\{\mathbf{x} \mapsto \mathbf{nil}, \mathbf{y} \mapsto \mathbf{a} : \mathbf{nil}\}, \{\mathbf{x} \mapsto \mathbf{a} : \mathbf{nil}, \mathbf{y} \mapsto \mathbf{nil}\}\}$. ■

The SLD semantics is not really what we are interested in for dataflow analysis purposes, because it does not associate information about runtime states with program points. The program points that we are interested in here are the "arrows" in a program. This is because we are primarily interested in specializing unification code. As in Section 3 we therefore extend our definition to one of the "sticky" SLD semantics. Our repository logs sets of atoms. Since we essentially want to partially evaluate unification code, we choose to let program points be the "arrows" in a program's clauses. The repository maps to each clause the set of atoms that the clause was called with. So the domain of repositories is $\mathbf{Rep} = \mathbf{Clause} \to \mathcal{P}\,\mathbf{Atom}$. A repository is created using the function $\mathbf{log} : \mathbf{Clause} \to \mathbf{Atom} \to \mathbf{Rep}$ defined by

$$\mathbf{log}\ \mathbf{C}\ \mathbf{A}\ \mathbf{C}' = \text{if } \mathbf{C} = \mathbf{C}' \text{ then } \{\mathbf{A}\} \text{ else } \emptyset.$$

Definition. The *sticky SLD semantics* has semantic function $\mathbf{O}' : \mathbf{Prog} \to \mathbf{Atom}{*} \to (\mathcal{P}\,\mathbf{Sub} \times \mathbf{Rep})$ and is defined as follows.

$$\begin{array}{lcl} \mathbf{O}'\ \mathbf{P}\ \mathbf{G} & = & \text{let } (\Theta, \mathbf{r}) = \mathbf{F}'_{\mathbf{P}}\ \mathbf{G}\ \iota \text{ in } (\Delta\ (\mathbf{restrict}\ (\mathbf{vars}\ \mathbf{G}))\ \Theta, \mathbf{r}) \\ \mathbf{F}'_{\mathbf{P}}\ \mathbf{nil}\ \theta & = & (\{\theta\}, \bot_{\mathbf{Rep}}) \\ \mathbf{F}'_{\mathbf{P}}\ (\mathbf{A} : \mathbf{G})\ \theta & = & \bigsqcup_{\mathbf{C} \in \mathbf{P}} \text{let } [\![\mathbf{H} \leftarrow \mathbf{B}]\!] = \mathbf{rename}\ ((\mathbf{vars}\ \theta) \cup (\mathbf{vars}\ (\mathbf{A} : \mathbf{G})))\ \mathbf{C} \text{ in} \\ & & \qquad \text{let } \Theta = \mathbf{mgu}\ \mathbf{A}\ \mathbf{H} \text{ in} \\ & & \qquad\qquad \bigsqcup_{\theta' \in \Theta}((\mathbf{F}'_{\mathbf{P}}\ (\theta'\ (\mathbf{B} :: \mathbf{G}))\ (\theta' \circ \theta)) \sqcup (\emptyset, \mathbf{log}\ \mathbf{C}\ \mathbf{A})). \end{array}$$ ■

This semantics is still not particularly suitable for dataflow analysis, as termination of the analyses cannot easily be guaranteed. Assume we are interested in determining which variables are bound to ground terms in calls to clauses. As descriptions we choose sets $\mathbf{U}$ of variables, with the intention that $\mathbf{V} \in \mathbf{U}$ means that the current substitution definitely grounds $\mathbf{V}$. Now consider the following program.

$$\mathbf{ancestor}(\mathbf{x},\mathbf{z}) \leftarrow \mathbf{parent}(\mathbf{x},\mathbf{z}).$$
$$\mathbf{ancestor}(\mathbf{x},\mathbf{z}) \leftarrow \mathbf{ancestor}(\mathbf{x},\mathbf{y}), \mathbf{ancestor}(\mathbf{y},\mathbf{z}).$$
$$\mathbf{parent}(\mathbf{a},\mathbf{b}).$$

Assume we are given the goal $\leftarrow\mathbf{ancestor}(\mathbf{a},\mathbf{z})$. An analysis based on the SLD semantics must compute an infinite number of goal/description pairs, namely (among others)

$$(\mathbf{ancestor}(\mathbf{x},\mathbf{z}), \{\mathrm{x}\})$$
$$(\mathbf{ancestor}(\mathbf{x},\mathbf{y}), \mathbf{ancestor}(\mathbf{y},\mathbf{z}), \{\mathrm{x}\})$$
$$(\mathbf{ancestor}(\mathbf{x},\mathbf{w}), \mathbf{ancestor}(\mathbf{w},\mathbf{y}), \mathbf{ancestor}(\mathbf{y},\mathbf{z}), \{\mathrm{x}\})$$
$$\vdots$$

Clearly such a dataflow analysis will not terminate. What we need is a semantic definition that somehow "merges" information about all incarnations of a variable that appears in a program. We develop such a definition in the following sections.

4.2 Parametric Substitutions

In this section we develop a theory of "substitutions" which differs somewhat from classical substitution theory. There are a number of reasons for doing this. First, classical substitutions have some drawbacks when used in a denotational definition. Most general unifiers are not unique, even when idempotent. For example, unifying $\mathbf{p}(\mathbf{x})$ and $\mathbf{p}(\mathbf{y})$, it is not clear whether the result should be $\{\mathbf{x} \mapsto \mathbf{y}\}$ or $\{\mathbf{y} \mapsto \mathbf{x}\}$, and it is hard to guarantee that a semantic definition is invariant under choice of unification function. Second, renaming traditionally causes problems. Existing approaches either ignore the problem by postulating some magical (invisible or nondeterministic) renaming operator, or they commit themselves to a particular renaming technique (as do Jones and Søndergaard [25]).

As an example of the problem, consider the program

$$\leftarrow\mathbf{p}(\mathbf{x}).$$
$$\mathbf{p}(\mathbf{x}).$$

By the definition of Jones and Søndergaard this program denotes $\{\{\mathbf{x} \mapsto \mathbf{x}_1\}\}$, but one could argue that $\{\{\mathbf{x} \mapsto \mathbf{x}_{17}\}\}$, $\{\{\mathbf{x} \mapsto \mathbf{z}\}\}$, or even $\{\iota\}$, would be just as appropriate (recall that ι is the identity substitution). It would be nice if substitutions would somehow "automatically" perform renaming, preferably in such a way that the definition made no assumptions about renaming technique. The urge to avoid use of some specific "renaming function" is so much more pressing as renaming is practically superfluous in the dataflow analyses that we aim to capture.

Our denotational definition is based on a notion of "parametric substitutions." These have previously been studied by Maher [29], although for a different purpose and under the name "parameterized substitutions." We follow Maher in distinguishing between *variables*, whose names are significant, and *parameters*, whose names are insignificant.

Recall that **Var** is a set of countably infinite *variables.* We think of **Var**, **Term**, and **Atom** as syntactic categories: the variables that appear in a program are all assumed to be taken from **Var**. In contradistinction to **Var**, **Par** is a countably infinite set of *parameters.* Both variables and parameters serve as "placeholders," but it proves useful to maintain the distinction: **Var** and **Par** are disjoint. A one-to-one correspondence between the two is given by the bijective function $\epsilon : \mathbf{Var} \rightarrow \mathbf{Par}$. In examples we distinguish variables from parameters by putting primes on the latter.

The set of terms over **Par** is denoted by $\mathbf{Term}[\mathbf{Par}]$, and the set of atoms over **Par** is denoted by $\mathbf{Atom}[\mathbf{Par}]$. A *substitution into* **Par** is a total mapping $\sigma : \mathbf{Var} \rightarrow \mathbf{Term}[\mathbf{Par}]$ which is "finitely based" in the sense that all but a finite number of variables are mapped into distinct parameters. We do not distinguish substitutions into **Par** from their natural extension to $\mathbf{Atom} \rightarrow \mathbf{Atom}[\mathbf{Par}]$. The set of substitutions into **Par** is denoted $\mathbf{Sub}[\mathbf{Par}]$.

Let us for the time being denote **Atom** by $\mathbf{Atom}[\mathbf{Var}]$ to stress its distinction from $\mathbf{Atom}[\mathbf{Par}]$, and let $\mathbf{X}$ be either **Var** or **Par**. We define the *standard preordering*, $\trianglelefteq$, on $\mathbf{Atom}[\mathbf{X}]$ by $\mathbf{A} \trianglelefteq \mathbf{A}'$ iff $\exists\, \tau : \mathbf{X} \rightarrow \mathbf{Term}[\mathbf{X}] \,.\, \mathbf{A} = (\tau\ \mathbf{A}')$. The standard preordering clearly induces an equivalence relation on $\mathbf{Atom}[\mathbf{Var}]$ (or $\mathbf{Atom}[\mathbf{Par}]$) which is "consistent variable (or parameter) renaming." We denote the resulting quotient by $\mathbf{Atom}[\mathbf{X}]_{\trianglelefteq}$ and use $\trianglelefteq$ to denote the induced partial ordering on $\mathbf{Atom}[\mathbf{X}]_{\trianglelefteq}$ as well. Note that $\mathbf{Atom}[\mathbf{X}]_{\trianglelefteq}$ has no least element.

For $\mathbf{A} \in \hat{\mathbf{A}}$, where $\hat{\mathbf{A}} \in \mathbf{Atom}[\mathbf{X}]_{\trianglelefteq}$, we may denote $\hat{\mathbf{A}}$ by $[\mathbf{A}]$. In fact we think of $[\cdot]$ as the function from $\mathbf{Atom}[\mathbf{X}]$ to $\mathbf{Atom}[\mathbf{X}]_{\trianglelefteq}$ that, given $\mathbf{A}$, yields the equivalence class of $\mathbf{A}$.

A *parametric substitution* is a mapping from **Atom** to $\mathbf{Atom}[\mathbf{Par}]_{\trianglelefteq}$. We will not allow all such mappings, though: the mappings we use are "essentially" substitutions. More precisely, the set $\mathbf{Psub} \subseteq \odot\, (\mathbf{Atom} \rightarrow \mathbf{Atom}[\mathbf{Par}]_{\trianglelefteq})$ of parametric substitutions is defined by

$$\mathbf{Psub} = \odot\ \{[\cdot] \circ \sigma \mid \sigma \in \mathbf{Sub}[\mathbf{Par}]\}.$$

That is, application of a parametric substitution can be seen as application of a substitution that has only parameters in its range, followed by taking the equivalence class of the result. We define $(\bot_{\mathbf{Psub}}\ \mathbf{A})$ to be the least element of $\odot\, \mathbf{Atom}[\mathbf{Par}]_{\trianglelefteq}$ and equip **Psub** with a partial ordering $\leq$, which is pointwise ordering on $\{[\cdot] \circ \sigma \mid \sigma \in \mathbf{Sub}[\mathbf{Par}]\}$.

Proposition 4.2 **Psub** *is a complete lattice.* ■

We use a notation for parametric substitutions similar to that used for substitutions; however, square rather than curly brackets are used, to distinguish the two. For example, $[\mathbf{x} \mapsto \mathbf{x}', \mathbf{y} \mapsto \mathbf{f}(\mathbf{x}')]$ will map $\mathbf{p}(\mathbf{x}, \mathbf{y}, \mathbf{z})$ to $[\mathbf{p}(\mathbf{x}', \mathbf{f}(\mathbf{x}'), \mathbf{z}')]$. This parametric substitution corresponds to the substitution $\{\mathbf{y} \mapsto \mathbf{f}(\mathbf{x})\}$.

Definition. The function $\mathbf{meet} : \mathcal{P}\,\mathbf{Psub} \to \mathcal{P}\,\mathbf{Psub} \to \mathcal{P}\,\mathbf{Psub}$ is defined by

$$\mathbf{meet}\ \Pi\ \Pi' = \{\pi \sqcap \pi' \mid \pi \in \Pi \wedge \pi' \in \Pi'\}.$$

The function $\mathbf{pmgu} : \mathbf{Atom} \to \mathbf{Atom}[\mathbf{Par}]_{\trianglelefteq} \to \mathbf{Psub}$ is defined by

$$\mathbf{pmgu}\ \mathbf{A}\ \hat{\mathbf{A}} = \bigsqcup\{\pi \mid \pi\ \mathbf{A} \trianglelefteq \hat{\mathbf{A}}\}.$$

The function $\mathbf{unify} : \mathbf{Atom} \to \mathbf{Atom} \to \mathcal{P}\,\mathbf{Psub} \to \mathcal{P}\,\mathbf{Psub}$ is defined by

$$\mathbf{unify}\ \mathbf{A}\ \mathbf{A}'\ \Pi = \{\mathbf{pmgu}\ \mathbf{A}\ (\pi\ \mathbf{A}') \mid \pi \in \Pi\}. \quad \blacksquare$$

Readers may wonder why our auxiliary functions are defined to operate on *sets* of parametric substitutions, when they will typically be applied only to singleton sets. In fact it is only in this section that they are used in that way. Later it proves useful to have the broader definitions.

Example 4.3 Let $\mathbf{A}_1 = \mathbf{p}(\mathbf{a}, \mathbf{x}), \mathbf{A}_2 = \mathbf{p}(\mathbf{y}, \mathbf{z})$, and let

$$\pi = [\mathbf{y} \mapsto \mathbf{y}', \mathbf{z} \mapsto \mathbf{f}(\mathbf{y}')].$$

Then $\mathbf{unify}\ \mathbf{A}_1\ \mathbf{A}_2\ \{\pi\} = \{[\mathbf{x} \mapsto \mathbf{f}(\mathbf{a})]\}$. Notice that this parametric substitution does not constrain $\mathbf{y}$ or $\mathbf{z}$, only variables in $\mathbf{A}_1$ may be constrained. On the other hand we have that $\mathbf{unify}\ \mathbf{A}_2\ \mathbf{A}_1\ \{\pi\} = \{[\mathbf{y} \mapsto \mathbf{a}]\}$, in which both $\mathbf{x}$ and $\mathbf{z}$ are unconstrained. Notice also that $\mathbf{unify}\ \mathbf{A}_1\ \mathbf{A}_1\ \{\pi\} = \{\epsilon\}$, while $\mathbf{unify}\ \mathbf{A}_2\ \mathbf{A}_2\ \{\pi\} = \{\pi\}$.

Let $\mathbf{A}_3 = \mathbf{p}(\mathbf{f}(\mathbf{y}), \mathbf{z})$. Then $\mathbf{unify}\ \mathbf{A}_3\ \mathbf{A}_2\ \pi = \{\pi\}$. There is no "occur check problem," because the names of placeholders in $(\pi\ \mathbf{A}_2)$ have no significance. One can think of this as automatic renaming being performed by **unify**.

Finally let $\mathbf{A}_4 = \mathbf{p}(\mathbf{x}, \mathbf{x})$. Then $\mathbf{unify}\ \mathbf{A}_4\ \mathbf{A}_2\ \{\pi\} = \{\bot_{\mathbf{Psub}}\}$, corresponding to failure of unification. $\blacksquare$

The idea behind **unify** should be clear from this example: not only does the function **unify** perform renaming "automatically," it also restricts interest to certain variables, namely those of its first argument.

Definition. The function $\beta : \mathbf{Sub} \to \mathbf{Psub}$ maps a substitution to the corresponding parametric substitution. It is defined by $\beta\ \theta\ \mathbf{A} = [\epsilon\ (\theta\ \mathbf{A})]$. $\blacksquare$

So every substitution θ has a corresponding parametric substitution $(\beta\ \theta)$. However, there are parametric substitutions that are not in the range of β. For example, there is no substitution θ such that $(\beta\ \theta) = [\mathbf{x} \mapsto \mathbf{f}(\mathbf{y}', \mathbf{z}')]$.

4.3 A Denotational Base Semantics

Definition. The *base semantics* has domain $\mathbf{Den} = \mathbf{Atom} \to \mathcal{P}\,\mathbf{Psub} \to \mathcal{P}\,\mathbf{Psub}$ and semantic functions

$$\begin{aligned} \mathbf{P_{bas}} &: \mathbf{Prog} \to \mathbf{Den} \\ \mathbf{C_{bas}} &: \mathbf{Clause} \to \mathbf{Den} \to \mathbf{Den}. \end{aligned}$$

It is defined as follows.

$$\begin{aligned} \mathbf{P_{bas}}\ \mathbf{P}\ \mathbf{A}\ \Pi &= \mathbf{d}_0\ \mathbf{A}\ \Pi \text{ where rec } \mathbf{d}_0 = \textstyle\bigsqcup_{\mathbf{C}\in\mathbf{P}} \mathbf{C_{bas}}\ \mathbf{C}\ \mathbf{d}_0 \\ \mathbf{C_{bas}}\ [\![\mathbf{H} \leftarrow \mathbf{B}]\!]\ \mathbf{d}\ \mathbf{A}\ \Pi &= \textstyle\bigcup_{\pi\in\Pi} \mathbf{meet}\ \{\pi\}\ (\mathbf{unify}\ \mathbf{A}\ \mathbf{H}\ (\Sigma\ \mathbf{d}\ \mathbf{B}\ (\mathbf{unify}\ \mathbf{H}\ \mathbf{A}\ \{\pi\}))). \quad \blacksquare \end{aligned}$$

This definition captures the essence of an SLD refutation-based interpreter using a standard computation rule and a parallel search rule. Recall that $(\Sigma\ \mathbf{d})$ is the sequentialized version of $\mathbf{d}$ (Σ was defined on page 3). The semantics is well-defined: $(\mathbf{C_{bas}}\ \mathbf{C})$ is clearly monotonic.

Note that finite failure is not distinguished from non-termination. For example, let $\mathbf{P}$ be the program consisting only of the clause $\mathbf{p}$. Let $\mathbf{P}'$ be the program consisting only of the clause $\mathbf{q} \leftarrow \mathbf{q}$, and consider the query $\leftarrow\mathbf{q}$. We have that $\mathbf{P_{bas}}\ \mathbf{P}\ \mathbf{q} = \mathbf{P_{bas}}\ \mathbf{P}'\ \mathbf{q} = \emptyset$. This is not a flaw in the semantic definition, it merely reflects the fact that we are not interested in the distinction, since termination issues are often disregarded in dataflow analysis. The statements we generate are of the form "whenever execution reaches this point, so and so holds." But in saying so, we do not actually say that execution does reach the point. In particular, "whenever the computation terminates, so and so holds" concludes nothing about termination.

Readers not familiar with the area should compare the above definition with others from the literature, both operational definitions, for example [5, 23, 28, 38] and denotational definitions, for example [13, 23, 25, 35]. Both the definition by Jones and Mycroft [23] and that by Debray and Mishra [13] assume a left-to-right computation rule and depth-first search of SLD trees and both capture non-termination. However, both use *sequences* of substitutions as denotations, which gives the semantics a rather different flavour. The definition by Jones and Søndergaard [25] uses sets of substitutions but is more complex than the present because of its use of an elaborate renaming technique. The present definition is closest to definitions previously suggested by Marriott and Søndergaard [35].

Let $\mathbf{restrict}\ \mathbf{U}\ \pi$ be the parametric substitution that acts like π when applied to variables in $\mathbf{U}$, but maps variables outside $\mathbf{U}$ to distinct parameters.

Theorem 4.4 $\theta \in \mathbf{O}\ \mathbf{P}\ \mathbf{A} \Rightarrow \mathbf{restrict}\ (\mathbf{vars}\ \mathbf{A})\ (\beta\ \theta) \in \mathbf{P_{bas}}\ \mathbf{P}\ \mathbf{A}\ \{\iota\}. \quad \blacksquare$

Again we make a detour to present a sticky semantics. As before, the repository keeps track of the atoms that clauses are called with, but for reasons that will soon be clear, the calling atom and the current substitution are kept separate. The domain of repositories is $\mathbf{Rep} = \mathbf{Clause} \to \mathcal{P}(\mathbf{Atom} \times \mathbf{Psub})$. A repository is created using the function $\mathbf{log} : \mathbf{Clause} \to (\mathbf{Atom} \times \mathbf{Psub}) \to \mathbf{Rep}$ defined by

$$\mathbf{log}\ \mathbf{C}\ \phi\ \mathbf{C}' = \text{if } \mathbf{C} = \mathbf{C}' \text{ then } \{\phi\} \text{ else } \emptyset.$$

Definition. The *sticky base semantics* has domain $\mathbf{Den} = \mathbf{Atom} \to \mathcal{P}\,\mathbf{Psub} \to ((\mathcal{P}\,\mathbf{Psub}) \times \mathbf{Rep})$ and semantic functions

$$\begin{aligned} \mathbf{P_{stk}} &: \mathbf{Prog} \to \mathbf{Den} \\ \mathbf{C_{stk}} &: \mathbf{Clause} \to \mathbf{Den} \to \mathbf{Den}. \end{aligned}$$

It is defined as follows.

$$\begin{aligned} \mathbf{P_{stk}}\ \mathbf{P}\ \mathbf{A}\ \Pi &= \mathbf{d}_0\ \mathbf{A}\ \Pi \text{ where rec } \mathbf{d}_0 = \textstyle\bigsqcup_{\mathbf{C} \in \mathbf{P}} \mathbf{C_{stk}}\ \mathbf{C}\ \mathbf{d}_0 \\ \mathbf{C_{stk}}\ [\![\mathbf{H} \leftarrow \mathbf{B}]\!]\ \mathbf{d}\ \mathbf{A}\ \Pi &= \textstyle\bigcup_{\pi \in \Pi} \text{let } \Pi' = \mathbf{unify}\ \mathbf{H}\ \mathbf{A}\ \{\pi\} \text{ in} \\ &\qquad (\mathbf{meet}\ \{\pi\}\ (\mathbf{unify}\ \mathbf{A}\ \mathbf{H}\ (\Sigma\ \mathbf{d}\ \mathbf{B}\ \Pi')), \mathbf{log}\ [\![\mathbf{H} \leftarrow \mathbf{B}]\!]\ (\mathbf{A}, \pi)). \quad \blacksquare \end{aligned}$$

A theorem about the relation between the sticky SLD semantics and the sticky base semantics, similar to Theorem 4.4, clearly holds.

Example 4.5 Let $\mathbf{P}$ be the program consisting of the two clauses (as before)

$$\begin{aligned} &\mathbf{C}_1 : \mathbf{append}(\mathbf{nil}, \mathbf{y}, \mathbf{y}). \\ &\mathbf{C}_2 : \mathbf{append}(\mathbf{u} : \mathbf{x}, \mathbf{y}, \mathbf{u} : \mathbf{z}) \leftarrow \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z}). \end{aligned}$$

and let $\mathbf{A}$ be the query $\leftarrow \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{a} : \mathbf{nil})$. We have that $\mathbf{P_{stk}}\ \mathbf{P}\ \mathbf{A}\ \{\epsilon\} = (\Pi, \mathbf{r})$, where

$$\begin{aligned} \Pi &= \{[\mathbf{x} \mapsto \mathbf{nil}, \mathbf{y} \mapsto \mathbf{a} : \mathbf{nil}], [\mathbf{x} \mapsto \mathbf{a} : \mathbf{nil}, \mathbf{y} \mapsto \mathbf{nil}]\} \\ \mathbf{r}\ \mathbf{C}_1 &= \{[\mathbf{y} \mapsto \mathbf{a} : \mathbf{nil}], [\mathbf{y} \mapsto \mathbf{nil}]\} \\ \mathbf{r}\ \mathbf{C}_2 &= \{[\mathbf{u} \mapsto \mathbf{a}, \mathbf{z} \mapsto \mathbf{nil}]\}. \quad \blacksquare \end{aligned}$$

The semantic function $\mathbf{P_{stk}}$ yields sufficient information for dataflow analyses, but is not in general finitely computable. This is the reason why we aim at realizing a variety of dataflow analyses by finding finitely computable approximations to $\mathbf{P_{stk}}$. These approximations must preserve the behavioural properties of "pure Prolog" programs. For instance, in applications to compiling, they should give reliable answers to questions such as "is this variable instantiated to a ground term?"

or "can this unification be done without the occur check?" Each such analysis requires the use of a coarser domain, the elements of which approximately describe substitutions or atom denotations.

Bye now it should be clear that the extension of the base semantics to a sticky semantics is fairly straightforward. To limit the complexity of definitions we shall forget about sticky semantics in the remainder of these notes. Readers should keep in mind, however, that ultimately the touchstone for the correctness of a dataflow analysis is its soundness with respect to the sticky semantics.

4.4 A Dataflow Semantics for Definite Logic Programs

The base semantics has been designed to capture the essence of SLD resolution. Now, however, we introduce some imprecision in the semantics. So far, for all substitutions generated by a clause, track was kept of the particular substitution the clause was called with, so that the "meet" of generated substitutions and call substitutions could be computed. We now abandon this approach in order to get closer to a dataflow semantics.

Definition. The *plural semantics* has domain $\mathbf{Den} = \mathbf{Atom} \to \mathcal{P}\,\mathbf{Psub} \to \mathcal{P}\,\mathbf{Psub}$ and semantic functions

$$\begin{aligned} \mathbf{P_{plu}} &: \mathbf{Prog} \to \mathbf{Den} \\ \mathbf{C_{plu}} &: \mathbf{Clause} \to \mathbf{Den} \to \mathbf{Den}. \end{aligned}$$

It is defined as follows.

$$\begin{aligned} \mathbf{P_{plu}}\ \mathbf{P}\ \mathbf{A}\ \Pi &= \mathbf{d}_0\ \mathbf{A}\ \Pi \text{ where rec } \mathbf{d}_0 = \textstyle\bigsqcup_{\mathbf{C} \in \mathbf{P}} \mathbf{C_{plu}}\ \mathbf{C}\ \mathbf{d}_0 \\ \mathbf{C_{plu}}\ [\![\mathbf{H} \leftarrow \mathbf{B}]\!]\ \mathbf{d}\ \mathbf{A}\ \Pi &= \mathbf{meet}\ \Pi\ (\mathbf{unify}\ \mathbf{A}\ \mathbf{H}\ (\Sigma\ \mathbf{d}\ \mathbf{B}\ (\mathbf{unify}\ \mathbf{H}\ \mathbf{A}\ \Pi))). \quad \blacksquare \end{aligned}$$

Proposition 4.6 $\mathbf{P_{plu}} \propto \mathbf{P_{bas}}$. $\blacksquare$

We now turn to dataflow semantics. To extract runtime properties of pure Prolog programs one can develop a variety of more or less abstract versions of the preceding semantics. To put many such (dataflow) semantic versions into a common framework, we extract from the plural semantics a *dataflow* semantics. This semantics contains exactly those features that are common to all the approximate versions that we want. It leaves one domain and two auxiliary functions unspecified. These missing details of the dataflow semantics are to be filled in by *interpretations* (see below). Different interpretations correspond to different dataflow analyses. The parametric domain, $\mathbf{X}$, contains whatever "descriptions" we choose in approximating sets of parametric substitutions. $\mathbf{X}$ should thus be a complete lattice which corresponds to $\mathcal{P}\,\mathbf{Psub}$ in the standard semantics in a way laid down by a Galois insertion $(\mathbf{X}, \gamma, \mathcal{P}\,\mathbf{Psub}, \alpha)$. The two auxiliary functions should "compose" and "unify" descriptions in a sound manner.

Definition. The *dataflow semantics* has domain $\mathbf{Den} = \mathbf{Atom} \to \mathbf{X} \to \mathbf{X}$ and semantic functions

$$\begin{aligned} \mathbf{P} &: \mathbf{Prog} \to \mathbf{Den} \\ \mathbf{C} &: \mathbf{Clause} \to \mathbf{Den} \to \mathbf{Den}. \end{aligned}$$

It is defined as follows.

$$\begin{aligned} \mathbf{P}\ \mathbf{P}\ \mathbf{A}\ \phi &= \mathbf{d}_0\ \mathbf{A}\ \phi \text{ where rec } \mathbf{d}_0 = \textstyle\bigsqcup_{\mathbf{C}\in\mathbf{P}} \mathbf{C}\ \mathbf{C}\ \mathbf{d}_0 \\ \mathbf{C}\ [\![\mathbf{H} \leftarrow \mathbf{B}]\!]\ \mathbf{d}\ \mathbf{A}\ \phi &= \mathbf{m}\ \phi\ (\mathbf{u}\ \mathbf{A}\ \mathbf{H}\ (\Sigma\ \mathbf{d}\ \mathbf{B}\ (\mathbf{u}\ \mathbf{H}\ \mathbf{A}\ \phi))). \quad ■ \end{aligned}$$

Note that the definition is incomplete because $\mathbf{X}$, $\mathbf{m}$, and $\mathbf{u}$ have not been specified. We assume that $\mathbf{X}$ is a complete lattice. The posets **Prog**, **Clause**, and **Atom** are ordered by identity, and the orderings on all other domains are then naturally induced by that on $\mathbf{X}$ (that is, all other domains are ordered pointwise). We now further overload the term "interpretation" to mean "that which fills in the missing details of the dataflow semantics."

Definition. An *interpretation* is a triple $(\mathbf{X}, \mathbf{m}, \mathbf{u})$ where $\mathbf{X}$ is a non-empty complete lattice, and $\mathbf{m} : \mathbf{X} \to \mathbf{X} \to \mathbf{X}$ and $\mathbf{u} : \mathbf{Atom} \to \mathbf{Atom} \to \mathbf{X} \to \mathbf{X}$ are (monotonic) functions. ■

We use the notational convention that $\mathbf{P_X}$ denotes the version of $\mathbf{P}$ which is obtained by completing the definition of the dataflow semantics with interpretation $\mathbf{I_X}$.

Proposition 4.7 *For every interpretation* $\mathbf{I_X}$*,* $\mathbf{P_X}$ *is well-defined.* ■

Definition. Let $\mathbf{I} = (\mathbf{X}, \mathbf{m}, \mathbf{u})$ and $\mathbf{I}' = (\mathbf{X}', \mathbf{m}', \mathbf{u}')$ be interpretations. Let $\mathbf{Den} = \mathbf{Atom} \to \mathbf{X} \to \mathbf{X}$ and $\mathbf{Den}' = \mathbf{Atom} \to \mathbf{X}' \to \mathbf{X}'$. Then $\mathbf{I}'$ is sound with respect to $\mathbf{I}$ under $\mathbf{G} = (\mathbf{X}', \gamma, \mathbf{X}, \alpha)$ iff $\mathbf{G}$ is a Galois insertion, $\mathbf{m}'$ **appr** $\mathbf{m}$, and $\mathbf{u}'$ **appr** $\mathbf{u}$. If there is a Galois insertion $\mathbf{G}$ such that $\mathbf{I}'$ is sound with respect to $\mathbf{I}$ under $\mathbf{G}$, we may simply say that $\mathbf{I}'$ is sound with respect to $\mathbf{I}$. ■

Theorem 4.8 *If interpretation* $\mathbf{I_x}$ *is sound with respect to* $\mathbf{I_y}$*, then* $\mathbf{P_x} \propto \mathbf{P_y}$ *holds.* ■

Clearly we obtain a partial ordering $\sqsubseteq$ on interpretations by defining $\mathbf{I} \sqsubseteq \mathbf{I}'$ iff $\mathbf{I}'$ is sound with respect to $\mathbf{I}$. This in fact gives rise to a complete lattice of semantics [7].

4.5 Applications

Approximating call patterns in logic programs seems very versatile. In the previous section we distilled a dataflow semantics which left undefined certain details assumed to be application dependent. The idea is that particular dataflow analyses can be obtained by providing interpretations that give the missing details. A large number of useful dataflow analyses fit into this framework.

Some dataflow analyses aim at providing information about the structure of the terms that will be generated at runtime. This is the case, for instance, for most analyses that would be characterized as "type analyses." For many purposes, however, it suffices to disregard term structure and functors altogether and use descriptions based entirely on the variables that appear in a program. This is the case for the groundness analysis detailed in Section 5 and many other useful analyses. Without claiming completeness of the list or the attached references, we mention some other applications that fit into the present or a very similar framework:

- Compile time garbage collection [6].
- Determinacy analysis [15].
- Floundering analysis for normal logic programs [10].
- Independence analysis for and-parallelism [18, 50].
- Intelligent backtracking [12].
- Mode analysis [5, 14, 38, 39].
- Occur check analysis [48].
- Program specialization and partial evaluation [16].
- Program transformation [34].
- Trailing analysis [49].
- Type inference [6, 19].

5 An Example: Groundness Analysis

In this section we give an example dataflow analysis for groundness propagation. The analysis is non-trivial and, to our knowledge, is more accurate than existing groundness analyses.

In this analysis, propositional formulas, or more precisely, classes of equivalent propositional formulas, are the descriptions. These descriptions are closely related to those used by Dart [9]. A parametric substitution π is described by a formula $\mathbf{F}$ if the truth assignment given by "$\mathbf{V}$ is true iff π grounds $\mathbf{V}$" satifies $\mathbf{F}$. For example, the formula $\mathbf{x} \leftrightarrow \mathbf{y}$ describes the parametric substitutions $[\mathbf{x} \mapsto \mathbf{a}, \mathbf{y} \mapsto \mathbf{b}]$ and $[\mathbf{x} \mapsto \mathbf{u}', \mathbf{y} \mapsto \mathbf{u}', \mathbf{u} \mapsto \mathbf{u}']$ but not $[\mathbf{x} \mapsto \mathbf{a}]$.

Let **Prop** be the poset of equivalent propositional formulas (ordered by implication) over some suitable finite variable set. We can represent an equivalence class in **Prop** by a canonical representative for the class, perhaps a formula which is in disjunctive or conjunctive normal form. By an abuse of notation we will apply logical connectives to both propositional formulas and to classes of equivalent formulas.

Lemma 5.1 *The poset* **Prop** *is a finite (complete) lattice with conjunction as the greatest lower bound and disjunction as the least upper bound.* ■

Definition. Let $\gamma : \mathbf{Prop} \to \mathcal{P}\,\mathbf{Psub}$ be defined by

$$\gamma\ \mathbf{F} = \{\pi \in \mathbf{Psub} \mid \mathbf{assign}\ \pi\ \mathbf{satisfies}\ \mathbf{F}\}$$

where $\mathbf{assign} : \mathbf{Psub} \to \mathbf{Var} \to \mathbf{Bool}$ is given by $\mathbf{assign}\ \pi\ \mathbf{V} = \mathbf{true}$ iff π grounds $\mathbf{V}$. We define $\alpha : \mathcal{P}\,\mathbf{Psub} \to \mathbf{Prop}$ by

$$\alpha\ \Pi = \sqcap\{\mathbf{F} \in \mathbf{Prop} \mid \Pi \subseteq \gamma\ \mathbf{F}\}. \quad \blacksquare$$

Example 5.2 Let $\mathbf{F} = [\mathbf{x} \leftrightarrow \mathbf{y}]$. Then $[\mathbf{x} \mapsto \mathbf{a}, \mathbf{y} \mapsto \mathbf{b}] \in \gamma\ \mathbf{F}$ and

$$\mathbf{F} = \alpha\ \{[\mathbf{x} \mapsto \mathbf{a}, \mathbf{y} \mapsto \mathbf{b}], [\mathbf{x} \mapsto \mathbf{u}', \mathbf{y} \mapsto \mathbf{u}']\}. \quad \blacksquare$$

Proposition 5.3 *The tuple* $(\mathbf{Prop}, \gamma, \mathcal{P}\,\mathbf{Psub}, \alpha)$ *is a Galois insertion.* $\blacksquare$

In the analysis the function **meet** is approximated by conjunction.

Definition. The function $\mathbf{meet}' : \mathbf{Prop} \to \mathbf{Prop} \to \mathbf{Prop}$ is defined by $\mathbf{meet}'\ \mathbf{F}\ \mathbf{F}' = \mathbf{F} \wedge \mathbf{F}'$. $\blacksquare$

It is straightforward to show that $\mathbf{meet}'$ is the best approximation to **meet**.

Proposition 5.4 $\mathbf{meet}' = \alpha \circ \mathbf{meet} \circ \gamma$. $\blacksquare$

The function **unify** is somewhat trickier to approximate. It makes use of **project**, a projection function on propositional formulas and **mgpu**, the analogue of **mgu** for parametric substitutions.

Definition. Let $\mathbf{tass}\ \mathbf{U}$ be the set of truth assignments to the variables in $\mathbf{U} \subseteq \mathbf{Var}$. Define the function $\mathbf{project} : \mathcal{P}\,\mathbf{Var} \to \mathbf{Prop} \to \mathbf{Prop}$ by

$$\mathbf{project}\ \mathbf{U}\ \mathbf{F} = \bigvee\{\psi\ \mathbf{F} \mid \psi \in \mathbf{tass}\ (\mathbf{vars}\ \mathbf{F}) \setminus \mathbf{W}\}.$$

Define the function $\mathbf{mgpu} : \mathbf{Atom} \to \mathbf{Atom} \to \mathbf{Psub}$ by

$$\mathbf{mgpu}\ \mathbf{A}\ \mathbf{A}' = \{\beta\ \theta \mid \theta \in \mathbf{mgu}\ \mathbf{A}\ \mathbf{A}'\}.$$

Define the function $\mathbf{unify}' : \mathbf{Atom} \to \mathbf{Atom} \to \mathbf{Prop} \to \mathbf{Prop}$ by

$$\begin{aligned}\mathbf{unify}'\ \mathbf{H}\ \mathbf{A}\ \mathbf{F} \ &= \ \text{let } (\mathbf{A}', \mathbf{F}') = \mathbf{rename}\ (\mathbf{vars}\ \mathbf{H})\ (\mathbf{A}, \mathbf{F}) \text{ in} \\ &\qquad\qquad \mathbf{project}\ (\mathbf{vars}\ \mathbf{H})\ (\mathbf{F} \wedge (\alpha\ (\mathbf{mgpu}\ \mathbf{H}\ \mathbf{A}))). \quad \blacksquare\end{aligned}$$

Example 5.5 Let $\mathbf{A} = \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z})$, $\mathbf{H} = \mathbf{append}(\mathbf{nil}, \mathbf{y}, \mathbf{y})$, and $\mathbf{H'} = \mathbf{append}(\mathbf{u} : \mathbf{x}, \mathbf{y}, \mathbf{u} : \mathbf{z})$. Then

$$\begin{array}{lcl}
\mathbf{unify'}\ \mathbf{H}\ \mathbf{A}\ [\mathbf{true}] & = & [\mathbf{true}] \\
\mathbf{unify'}\ \mathbf{A}\ \mathbf{H}\ [\mathbf{true}] & = & [\mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{z})] \\
\mathbf{unify'}\ \mathbf{H'}\ \mathbf{A}\ [\mathbf{true}] & = & [\mathbf{true}] \\
\mathbf{unify'}\ \mathbf{A}\ \mathbf{H'}\ [\mathbf{false}] & = & [\mathbf{false}] \\
\mathbf{unify'}\ \mathbf{A}\ \mathbf{H'}\ [\mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{z})] & = & [(\mathbf{x} \wedge \mathbf{y}) \leftrightarrow \mathbf{z}]. \quad \blacksquare
\end{array}$$

Proposition 5.6 $\mathbf{unify'} = \alpha \circ \mathbf{unify} \circ \gamma$. $\blacksquare$

The *groundness analysis* $\mathbf{P_{ground}}$ is given by instantiating the dataflow semantics with the interpretation $(\mathbf{Prop}, \mathbf{meet'}, \mathbf{unify'})$.

Example 5.7 Let $\mathbf{P}$ be the **append** program

$\mathbf{append}(\mathbf{nil}, \mathbf{y}, \mathbf{y}).$
$\mathbf{append}(\mathbf{u} : \mathbf{x}, \mathbf{y}, \mathbf{u} : \mathbf{z}) \leftarrow \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z}).$

Consider the goal $\mathbf{A} = \leftarrow\mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z})$. To compute $\mathbf{P_{ground}}\ \mathbf{P}\ \mathbf{A}\ [\mathbf{true}]$ the analysis proceeds as follows. Let $\mathbf{f}\ \mathbf{d} = \bigsqcup_{\mathbf{C} \in \mathbf{P}} \mathbf{C}\ \mathbf{C}\ \mathbf{d}$. We then have

$$\begin{array}{lclclcl}
(\mathbf{f} \uparrow 0)\ \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z})\ [\mathbf{true}] & = & [\mathbf{false}] & & \\
(\mathbf{f} \uparrow 1)\ \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z})\ [\mathbf{true}] & = & [\mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{z})] \vee [\mathbf{false}] & = & [\mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{z})] \\
(\mathbf{f} \uparrow 2)\ \mathbf{append}(\mathbf{x}, \mathbf{y}, \mathbf{z})\ [\mathbf{true}] & = & [\mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{z})] \vee [(\mathbf{x} \wedge \mathbf{y}) \leftrightarrow \mathbf{z}] & = & [(\mathbf{x} \wedge \mathbf{y}) \leftrightarrow \mathbf{z}]
\end{array}$$

and $\mathbf{f} \uparrow 3 = \mathbf{f} \uparrow 2 = \mathbf{d}_0$. Thus $\mathbf{P_{ground}}\ \mathbf{P}\ \mathbf{A}\ [\mathbf{true}] = [(\mathbf{x} \wedge \mathbf{y}) \leftrightarrow \mathbf{z}]$.

Similarly one can show that $\mathbf{P_{ground}}\ \mathbf{P}\ \mathbf{A}\ [\mathbf{x} \wedge \neg\mathbf{y}] = [\mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{z})]$. In the sticky version of $\mathbf{P_{ground}}$, we would find that in all calls to **append**, the first argument is definitely ground, and the second argument is definitely not ground. $\blacksquare$

Theorem 5.8 $\mathbf{P_{ground}} \propto \mathbf{P_{plu}}$. $\blacksquare$

When performing a dataflow analysis, one is usually interested in computing the value of $(\mathbf{P_{ground}}\ \mathbf{P})$ for some fixed query consisting of an atom $\mathbf{A}$ and propositional formula $\mathbf{F}$. To do this efficiently one can make use of techniques for efficient fixpoint computation. The first thing to observe is that one is not interested in computing $(\mathbf{P_{ground}}\ \mathbf{P})$ for all queries. The idea is to push information about the query into the fixpoint computation and only compute those values of $(\mathbf{P_{ground}}\ \mathbf{P})$ actually required. This can be done by simultaneously computing the set of queries which may be called and the denotation of these queries. Standard tabulation techniques can be used for this [11]. The same basic idea appears in the magic set transformation used in deductive databases [3]. It is also related to the "minimal function graph" semantics introduced by Jones and Mycroft [24]. See also Marriott and Søndergaard [35].

Analysis efficiency may also be improved by first computing minimal cliques of mutually recursive clauses and performing the analysis on these (relatively) independently. One may also be able to make use of differential fixpoint techniques. In the groundness analysis this is facilitated if the classes of propositional formulas are represented by formulas in disjunctive normal form.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] K. Apt and M. van Emden. Contributions to the theory of logic programming. *Journal of the ACM* **29** : 841–862, 1982.

[3] F. Bancilhon *et al.* Magic sets and other strange ways to implement logic programs. In *Proc. Fifth ACM Symp. Principles of Database Systems*, pages 1–15. Cambridge, Massachusetts, 1986.

[4] G. Birkhoff. *Lattice Theory* (AMS Coll. Publ. XXV). American Mathematical Society, third edition 1973.

[5] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. To appear in *Journal of Logic Programming.*

[6] M. Bruynooghe *et al.* Abstract interpretation: towards the global optimization of Prolog programs. In *Proc. Fourth Int. Symp. Logic Programming*, pages 192–204. San Francisco, California, 1987.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Fourth Ann. ACM Symp. Principles of Programming Languages*, pages 238–252. Los Angeles, California, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Sixth Ann. ACM Symp. Principles of Programming Languages*, pages 269–282. San Antonio, Texas, 1979.

[9] P. Dart. On derived dependencies and connected databases. To appear in *Journal of Logic Programming.*

[10] P. Dart, K. Marriott and H. Søndergaard. Detection of floundering in logic programs. In preparation. Dept. of Computer Science, University of Melbourne, Australia.

[11] S. K. Debray. Efficient dataflow analysis of logic programs. In *Proc. Fifteenth Ann. ACM Symp. Principles of Programming Languages*, pages 260–273. San Diego, California, 1988.

[12] S. K. Debray. *Global Optimization of Logic Programs.* Ph. D. Thesis, State University of New York at Stony Brook, New York, 1986.

[13] S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming* **5** (1) : 61–91, 1988.

[14] S. K. Debray and D. S. Warren. Automatic mode inference of logic programs. *Journal of Logic Programming* **5** (3) : 207–229, 1988.

[15] S. K. Debray and D. S. Warren. Detection and optimization of functional computations in Prolog. In E. Shapiro, editor, *Proc. Third Int. Conf. Logic Programming* (Lecture Notes in Computer Science 225), pages 490–504. Springer-Verlag, 1986.

[16] J. Gallagher, M. Codish and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing* **6** (2,3) : 159–186, 1988.

[17] M. Hecht. *Flow Analysis of Computer Programs.* North-Holland, 1977.

[18] D. Jacobs and A. Langen. Static analysis of logic programs for independent and-parallelism. Technical Report 89-03, Computer Science Dept., University of Southern California, Los Angeles, California, 1989.

[19] G. Janssens and M. Bruynooghe. An application of abstract interpretation: Integrated type and mode inferencing. Report CW 86, Dept. of Computer Science, University of Leuven, Belgium, 1989.

[20] J. Jensen. Generation of machine code in Algol compilers. *BIT* **5** : 235–245, 1965.

[21] N. D. Jones. Flow analysis of lambda expressions. In S. Even and O. Kariv, editors, *Proc. Eighth Int. Coll. Automata, Languages and Programming* (Lecture Notes in Computer Science 115), pages 114–128. Springer-Verlag, 1981.

[22] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, pages 102–131. Prentice-Hall, 1981.

[23] N. D. Jones and A. Mycroft. A stepwise development of operational and denotational semantics for Prolog. In *Proc. 1984 Int. Symp. Logic Programming*, pages 289–298. Atlantic City, New Jersey, 1984.

[24] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proc. Thirteenth Ann. ACM Symp. Principles of Programming Languages*, pages 296–306. St. Petersburg, Florida, 1986.

[25] N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In Abramsky and Hankin [1], pages 123–142.

[26] T. Kanamori and T. Kawamura. Analyzing success patterns of logic programs by abstract hybrid interpretation. ICOT TR-279, ICOT, Tokyo, Japan, 1987.

[27] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, 1971. Originally published by Van Nostrand, 1952.

[28] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition 1987.

[29] M. Maher. On parameterized substitutions. Unpublished manuscript. IBM T. J. Watson Research Center, Yorktown Heights, New York, 1986.

[30] K. Marriott. Generalizing abstract interpretation. In preparation. IBM T. J. Watson Research Center, Yorktown Heights, New York, 1989.

[31] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 733–748. MIT Press, 1988.

[32] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. Research Report RC 14858, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1989.

[33] K. Marriott and H. Søndergaard. A fully abstract semantics for Horn clause programs. In preparation. IBM T. J. Watson Research Center, Yorktown Heights, New York, 1989.

[34] K. Marriott and H. Søndergaard. Prolog program transformation by introduction of difference-lists. In *Proc. Int. Computer Science Conf. 88*, pages 206–213. IEEE Computer Society, Hong Kong, 1988.

[35] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. X. Ritter, editor, *Information Processing 89*, pages 601–606. North-Holland, 1989.

[36] K. Marriott, H. Søndergaard and N. D. Jones. Denotational abstract interpretation of pure Prolog. In preparation. Dept. of Computer Science, University of Melbourne, Australia, 1989.

[37] C. S. Mellish. The automatic generation of mode declarations for Prolog programs. DAI Research Paper No. 163, University of Edinburgh, Scotland, 1981.

[38] C. S. Mellish. Abstract interpretation of Prolog programs. In E. Shapiro, editor, *Proc. Third Int. Conf. Logic Programming* (Lecture Notes in Computer Science 240), pages463–474. Springer-Verlag 1986.

[39] C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming* **2** (1) : 43–66, 1985.

[40] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph. D. Thesis, University of Edinburgh, Scotland, 1981.

[41] A. Mycroft and N.D. Jones. A relational framework for abstract interpretation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects* (Lecture Notes in Computer Science 217), pages 536–547. Springer-Verlag, 1986.

[42] P. Naur. The design of the Gier Algol compiler, part II. *BIT* **3** : 145–166, 1963.

[43] F. Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation* **76** (1) : 29–92, 1988.

[44] J. C. Reynolds. Automatic computation of data set definitions. In A. Morrell, editor, *Information Processing 68*, pages 456–461. North-Holland, 1969.

[45] J. C. Reynolds. On the relation between direct and continuation semantics. In J. Loeckx, editor, *Proc. Second Int. Coll. Automata, Languages and Programming* (Lecture Notes in Computer Science 14), pages 141–156. Springer-Verlag, 1974.

[46] D. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[47] M. Sintzoff. Calculating Properties of Programs by Valuation on Specific Models. *SIGPLAN Notices* **7** (1) : 203–207, 1972. Proc. ACM Conf. Proving Assertions about Programs.

[48] H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86* (Lecture Notes in Computer Science 213), pages 327–338. Springer-Verlag, 1986.

[49] A. Taylor. Removal of dereferencing and trailing in Prolog compilation. In G. Levi and M. Martelli, editors, *Logic Programming: Proc. Sixth Int. Conf.*, pages 48–60. MIT Press, 1989.

[50] W. Winsborough and A. Wærn. Transparent and-parallelism in the presence of shared free variables. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pages 749–764. MIT Press, 1988.