An abstract model for constraint logic programming languages

Niels Jørgensen
Roskilde University

An abstract model for constraint logic programming languages

Niels Jørgensen

Computer Science Dept., Roskilde University
P.O. Box 260, DK-4000 Roskilde, Denmark
`nielsj@dat.ruc.dk`
phone: + 45 46 75 77 11
fax: + 45 46 75 42 01

Abstract

The model CLP($\mathcal{X}$)$_{pat}$ is proposed as a tool for a certain class of dataflow analyses of CLP languages: analyses which are defined in terms of *call and success patterns*. The model is defined using Marriott, Søndergaard, and Jones' [20] theory of denotational abstract interpretation of logic programs. In the model's semantic definition for standard interpretation of CLP languages, the input and output of clauses consists of sets of constrained atoms, that is, of atom-constraint pairs. In the non-standard interpretation of a CLP program, a call or success pattern is a description of such a set. Two well-known groundness analyses are re-formulated within the framework. The analyses use propositional formulas to describe the groundness of program variables (cf. [18, 4]). The simplest one uses formulas that are conjunctions of variables, and the more refined is the *pos* analysis which employs arbitrary, so-called positive propositional formulas.

It can be observed that while many early analyses of Prolog were stated in terms of call and success patterns, the use of patterns has declined in recent years, especially in research were emphasis is on formal justification of correctness. On the basis of the two example analyses, we argue that call and success patterns may provide a venue to more intuitive analyses, while retaining a sound, formal basis, and so perhaps they deserve a come-back into the field of abstract interpretation of Prolog and other CLP languages.

Contents

1 Introduction **2**
1.1 Contributions . . . 3
1.2 Informal sketch of pattern-based CLP analysis . . . 4
1.3 Related work . . . 6

2 Syntax and informal operational semantics of a family of constraint logic languages **8**
2.1 Syntax . . . 8
2.2 Renaming . . . 9
2.3 Informal operational semantics . . . 10
2.4 Lattice theory and fixpoints . . . 11

3 Standard interpretation in CLP($\mathcal{X}$)$_{pat}$ **11**
3.1 The standard top-down semantics $\mathbf{P}^\sharp$ . . . 11
3.2 The primitive operators . . . 13
3.3 Correctness via Marriott and Søndergaard's semantics . . . 14

4 Non-standard interpretation in CLP($\mathcal{X}$)$_{pat}$ **16**

5 Groundness analysis of Prolog using propositional formulas **19**
5.1 Preliminaries . . . 19
5.2 Constraint analyses instances *conj* and *pos* . . . 21

6 Patterns for a simple groundness analysis: ***conj*** **23**

7 Patterns for a very precise groundness analysis: ***pos*** **27**

8 Conclusion **32**

A The equivalence of $\mathbf{P}_{van}$ and $\mathbf{P}^\sharp$ **34**

B Correctness of ***conj*** **35**

C Correctness of ***pos*** **36**

1 Introduction

The model CLP($\mathcal{X}$)$_{pat}$ is intended to serve as a basis for abstract interpretation of constraint logic programming languages — including Prolog. The X is meant to indicate that the model is parameterized with respect to the domain of constraint solving. The subscript pat indicates that the model is tailored for dataflow analyses that operate on domains of call and success patterns. In general, the model is abstract in the sense that it is intended to capture language properties of importance for abstract interpretation, and disregard other properties.

The model is proposed in an attempt to answer two questions:

- Is it possible to provide a *rigorous, formal basis* — in accordance with the tradition originating from Cousot and Cousot's theory of approximate computation [5] — for dataflow analyses of CLP languages which are defined in terms of call and success patterns ?

- Is it *useful* to define CLP dataflow analyses in terms of call and success patterns ?

In the groundness analysis to be formalized in the paper, an example call pattern is $p(\mathsf{G}, \mathsf{A})$. The pattern represents all calls to the binary predicate p in which the first argument is ground, and the second argument is "any", that is, nothing is known about it. The distinguishing feature of analyses based on call and success patterns is, in our view, the *symmetry of the input and output* of clauses. Thus, output from the clauses defining p is described by success patterns drawn from the same domain as the call pattern, say $p(\mathsf{G}, \mathsf{G})$. Patterns may be of a more fine-grained nature, say $p(f(\mathsf{A}, \mathsf{A}), \mathsf{G})$ representing input or output where the first argument is a term with the binary function symbol 'f'. Also, by the approach taken here, patterns may describe certain dependencies. For instance we will define an analysis where $\langle rev([H|T], Rev), (H \wedge T) \leftrightarrow Rev \rangle$ expresses that in atom $rev([H|T], Rev)$, variable Rev is ground if and only if variables H and T are ground. In lack of a better terminology, we refer to this approach as "pattern-oriented". Aside from the goal of attaining a rigorous framework, a secondary goal is to attain a *generic* framework — one that makes the fewest possible assumptions about the nature of the specific analyses that can be defined within it. This is to render the framework useful for a wide range of analyses - say, producing groundness or freeness information.

Figure 1 indicates that the model CLP($\mathcal{X}$)$_{pat}$ consists of a top-down semantic function $\mathbf{P}^\sharp$, which is intended to capture aspects of the language's operational

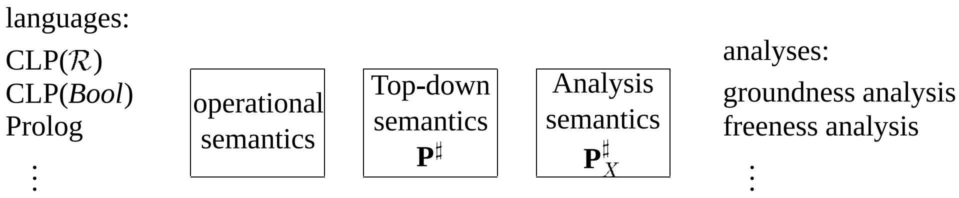

Figure 1: Overview of the model CLP($\mathcal{X}$)$_{pat}$.

semantics of interest for the analysis. Further, the model includes a format for defining a specific analysis instance X, and an associated analysis semantics $\mathbf{P}^\sharp_X$ which is parameterized with respect to analysis instance. A dataflow analysis is obtained by plugging in a specific analysis instance.

1.1 Contributions

The paper's first contribution is the formalization of the idea that a call or success pattern represents a set of pairs, each pair consisting of an atom and a constraint. (We call such a pair a *constrained atom*). To attain a rigorous, formal framework — and so answer the first question in the affirmative — the model CLP($\mathcal{X}$)$_{pat}$ is defined using Marriott, Søndergaard, and Jones' theory of *denotational abstract interpretation* of logic programs [20]. The theory provides the model's approach to the problem of ensuring, once and for all, that if a specific analysis instance X satisfies certain requirements, then the analysis semantics $\mathbf{P}^\sharp_X$ is guaranteed to be correct, in the sense of producing true statements about the programs subjected to analysis. The formal framework also differs from previously proposed ones in employing a simplified form of renaming: a fixed variable-to-variable function is used to avoid name-clashed for variables in calling atoms and clause heads, rather than a renaming function that depends on the given atom/head pair. Finally, the framework suggests that analyses are developed in two phases, firstly by specifying a domain of abstract constraints (and an associated operator representing constraint conjunction), and secondly by specifying a domain of patterns (and two associated operators).

The paper's second contribution is the examination of whether two well-known variants of Prolog groundness analysis using propositional formulas may benefit from using the pattern-oriented approach. The two analyses are referred to here as *conj* and *pos*, for conjunctions and so-called positive formulas. The two analyses vary greatly in the degree of analysis precision they provide (and in their computational cost). Each of the analyses are defined in two phases, as required by the framework: In a first phase, the use of propositional formulas to describe sets of *constraints* in a groundness analysis is formalized. (This is based on work on by Marriott and Søndergaard [18] and Cortesi *et al.* [4]). In a second phase, a domain of *patterns* is developed, with the objective that the domain fits the specific domain of propositional formulas. For the simplest domain, that of formulas that are conjunctions of propositional variables, the conclusion is that the pattern-oriented approach provides an intuitively very understandable analysis, and at the same time an elegant, formal justification for avoiding certain redundant computations. For the advanced domain, that of positive formulas, it is well-known that it is preferable to embed the analysis in a query-independent (bottom-up) context, because this improves efficiency at no loss of analysis precision. We present a new query-independent realization of groundness analysis using *pos*. By the query-independent approach, the analysis does not involve call patterns, only success patterns, and we argue that this provides the most natural representation for this very precise analysis. All in all, the answer to the second question is: Yes, sometimes patterns are indeed useful.

1.2 Informal sketch of pattern-based CLP analysis

The main advantage of working with call patterns is perhaps that they provide a formal basis for identifying calls that are identical except for the names of variables. In analysis of (constraint) logic programs, the names of variables is not part of the information sought; we may want to know whether a variable X is ground or free *etc.*, but we don't care about, say the names of the variables it may possibly be bound to. Consider the following program fragment for *quicksort* in Prolog, subjected to a groundness analysis using propositional formulas, so that the formula $H \wedge T$ represents all constraints that ground variables H and T (see also Sections 5–6). In the example, it is assumed that upon entry of the clause, it is known that H and T are ground. The conjunctions listed as comments (following the character %) indicate that more and more variables are inferred to be ground, as the atoms in the clause body are evaluated.

```
qs([], []).
qs([H|T], Sort) :-                 % H ∧ T
  split(H, T, Sm, Bi),             % H ∧ T ∧ Sm ∧ Bi
  qs(Sm, SortSm),                  % H ∧ T ∧ Sm ∧ Bi ∧ SortSm
  qs(Bi, SortBi),                  % ...
  app(SortSm, [H|SortBi], Sort).
```

Consider the atoms $qs(Sm, SortSm)$ and $qs(Bi, SortBi)$, in view of the assumption that the variables Sm and Bi are ground at the time of these two recursive calls to qs. Then it is clearly useful to view these calls as being equivalent, so that when the first call has been evaluated, grounding $SortSm$, this result can be used to infer that the second call grounds $SortBi$. In the pattern-oriented approach, we may use a call pattern such as $qs(\mathsf{G}, \mathsf{A})$ to describe all calls wherein the first argument is ground, thus accounting for both recursive calls. Evaluating the call pattern then produces the success pattern $qs(\mathsf{G}, \mathsf{G})$ which provides the answer for both calls. For an overview, see Figure 2.

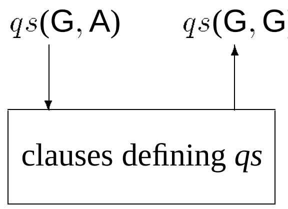

Figure 2: Abstract input and output of clauses for qs.

Corresponding to the use of patterns in the abstract interpretation of programs, in the concrete interpretation represented by the model's $\mathbf{P}^\sharp$ input and output to and from clauses are (sets of) constrained atoms. Suppose the *quicksort* program is called from a clause body

$$\ldots, qs(L, S), \ldots$$

at a point where the accumulated constraint is $L = [2, 1]$. Then input to the *quicksort* clauses is the constrained atom shown in Figure 3. In generel, in constrained atom $\langle A, \pi \rangle$ constituting input to a clause, atom A in constrained atom is always

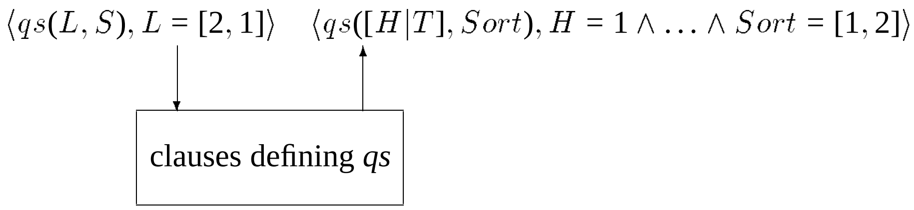

Figure 3: Concrete input and output of clauses for qs.

the atom that actually generates the call. Output is also a constrained atom; this atom being the head of the clause used to resolve the call.

The reader may have observed that the output constrained atom of Figure 3 is in terms of the variables in the program clause. This is due to a technique of renaming queries away from clauses (cf. [20]), rather than the other way around, as usual. The approach is orthogonal to the use of call/success patterns, yet particularly appropriate here, in view of the goal of simplifying the treatment of variable renaming. The approach allows for the analysis to operate on the original variant of a program clause, as indicated by the "comment constraints" $H \wedge T$ *etc.*. The idea is that this may yield a more understandable analysis than if the variables are renamed into $X_1, X_2, \ldots$ *etc.*

1.3 Related work

As an alternative to patterns, in a CLP analysis one may view clause-input as a calling atom in conjunction with an abstract constraint, and clause-output as another abstract constraint. We refer to this approach as "constraint-oriented". (At this point, an abstract constraint is an entity that describes a set of constraints). This view dominates the field of abstract interpretation of CLP languages. Similarly, analyses of Prolog most frequently view the input to clauses as constituted by a calling atom plus an abstract substitution, producing an output abstract substitution. Theories and frameworks that take this approach include (for CLP) Marriott and Søndergaard's [17], Garcia de la Banda and Hermenegildo's [7], and Bruynooghe and Boulanger's [3], and in the Prolog context the works [2, 16, 26, 20]. Indeed, the constraint-oriented approach is in a sense more natural: it is consistent with viewing CLP program execution as constraint manipulation, and analogously, for Prolog the manipulation of substitutions.

In spite of this, many suggestions for dataflow analysis of Prolog have been stated in terms of call and success patterns. Historically, the approach can be traced to D.H.D. Warren's work on Prolog compilation in the mid-seventies [25]. Warren introduced mode declarations as an option that allowed the programmer to express that a predicate would always be called in a certain way. The compiler would then produce improved code for that particular predicate. Mellish [21, 22] discussed a dataflow analysis operating on a set of modes which was an extension of those considered by Warren. The starting point for this analysis was a collection of user-provided mode declarations, and the purpose of the analysis would then be to infer modes for all predicates in the given program. Another early work was Mannila and Ukkonen's [14]. More recently proposed Prolog analyses phrased in terms of call and success patterns include the sophisticated domains of the analyser used with the Aquarius Prolog system [8]. However, the use of call/success patterns seems to have declined in recent years. Indeed, to obtain a formal framework for a pattern-based analysis, one must resort to one of the above-mentioned constraint-oriented frameworks, and then one is faced by a gap between the theory and the actual analysis.

Related work also includes Debray's analysis framework for Prolog [6]. The analysis employs call and success patterns, and provides information about the groundness and freeness of variables, as well as some information about dependency between variables. Kanamori [12] considers a groundness analysis similar to the analysis defined here in Section 6 (cf. the propositional domain *conj*). These two analyses are examples of analyses that are justified by formal correctness proofs, as well as shown to be useful, and so answer both our initial questions in the affirmative. The main difference to the present work is that the model CLP($\mathcal{X}$)$_{pat}$ is generic in the sense of being independent of a specific analysis. In fact, the two analyses could be restated as analysis instances of the model. Bruynooghe's framework for Prolog [2] was listed above among the analyses that are of a substitution-oriented nature. The framework introduces the notions of call and success specifications, these notions bearing resemblance to the notions of call and success patterns as defined here. However, we view the framework's overall approach as being substitution-oriented, because there is only one abstract domain: a generic domain of abstract substitutions.

The semantic definition for CLP given in my thesis [11] is in some respects similar to the definition of the top-down semantics $\mathbf{P}^\sharp$ given here. The main difference is that the present definition can serve as a basis for analyses of a more precise nature, because the process of variable renaming is modelled in a more fine-grained manner. This is reflected in the proof of equivalence with a (relatively fine-grained) semantic definition given by Marriott and Søndergaard.

The remainder of the paper is organized as follows:

Section 2 defines a family of CLP languages, *i.e.*, the scope of the model CLP($\mathcal{X}$)$_{pat}$. Sections 3–4 then define the model CLP($\mathcal{X}$)$_{pat}$: the top-down semantics $\mathbf{P}^\sharp$ (Sect. 3), and the notions of constraint analysis instance, analysis instance, and analysis semantics (Sect. 4). Section 5 defines two constraint analysis instances — *conj* and *pos* — for Prolog groundness analysis, using propositional formulas to describe sets of constrains. Sections 6–7 examine whether it is useful to add domains of patterns to these two well-known groundness analyses. Finally, Section 8 concludes.

2 Syntax and informal operational semantics of a family of constraint logic languages

The model CLP($\mathcal{X}$)$_{pat}$ is intended to capture properties of a given constraint logic language CLP($\mathcal{X}$), for instance CLP($\mathcal{R}$) or Prolog. It is the intention that our definition of syntax and informal operational semantics of CLP($\mathcal{X}$) is consistent with the language model defined by Jaffar and Lassez in [9].

We follow [20] in finding that Prolog be best analyzed as a CLP language. This implies that Prolog's unification is represented by formation of equality constraints. For instance, the result of unifying calling atom $p(a, X)$ with clause head $p(Y, Z)$ is represented by the equation $p(a, X) = p(Y, Z)$, instead of a most general unifier, say $\{Y \mapsto a, Z \mapsto X\}$. Working with equations is a more abstract approach. This is in the sense that one disregards the information, inherent in substitutions, of specific solutions to equations. The more abstract approach is appropriate since equations are simpler to deal with than substitutions, while still accounting for all information sought in, say a groundness analysis.

2.1 Syntax

A CLP($\mathcal{X}$) program $P \in Progr$ is a collection of clauses $C \in Clause$ built from the syntactical categories $Atom$, Con, Var, $Term$, Fct, and $Pred$. In clause $C = H :- B$ the clause head H is an atom, and the body B is a (possibly empty) sequence of atomic formulas. An atomic formula is either an atom or a constraint. An atom $p(T_1, \ldots, T_n) \in Atom$ is build from an n-ary predicate symbol $p \in Pred$ and n terms. A term $T \in Term$ is build in the usual way from variables (from Var) and function symbols (from Fct). A primitive constraint $\pi \in Con$ has the form $c(T_1, \ldots, T_n)$ where c is an n-ary predicate symbol and the arguments are terms. One predicate symbol is the binary equality symbol "=". *true* and *false* are also primitive constraints. Con is closed under conjunction, existential quantification, and variable renaming. An occurrence of a variable in a constraint is free if it is not within the scope of an existential quantifier over the variable. We write $vars\ A$ and $vars\ \pi$ for the set of variables occurring in atom A and constraint π, respectively. Variable V belongs to $vars\ \pi$ if there is an occurrence of V in π, independently of whether the occurrence is free. Let $W \subseteq Var$. We write $\exists_{\overline{W}}.\pi$ for the constraint obtained by quantifying existentially all the free variables of π, except for those in W. Example programs are written in accordance with the Edinburgh syntax.

For simplicity we do not make any assumptions about the semantics of constraints, that is, the way constraints are interpreted with respect to an underlying domain of constraint solving, say the domain of real numbers in the case of CLP($\mathcal{R}$). However, we do assume that any constraint is either satisfiable in the underlying constraint structure, or not. Satisfiability is represented by a unary relation sat on Con, where sat(*true*) holds, while sat(*false*) does not.

2.2 Renaming

The set of variables Var is divided into two disjoint and countably infinite sets: $Var = Var_{pg} \cup Var_{rn}$. Variables in programs and queries must belong to Var_{pg}, while Var_{rn} is used as a reservoir that supplies variables for the use in renaming. It is assumed that Var_{rn} is divided into a infinite collection of disjoint, infinite subsets $Var_1, Var_2, \ldots$. The definition of the semantics $\mathbf{P}^\sharp$ in Section 3 makes use of a fixed renaming $\rho \in Var \rightarrow Var$. ρ is assumed to be injective (and total), with the image $\rho(Var_{pg})$ of Var_{pg} under ρ being Var_1, and for all i, $\rho(Var_i) = Var_{i+1}$. Renaming ρ is identified with it's natural extension to atoms and constraints. For constraints, renaming extends also to variables that are existentially quantified, so that for instance $\rho(\exists Y.\pi(X, Y)) = \exists(\rho Y).\pi(\rho X, \rho Y)$.

The use of ρ hinges on the fact that in the semantic definition, queries are renamed away from clauses. Suppose we are given an atom A whose variables are program variables, and that we want to rename atom H and constraint π away from those program variables. Then ρ is a valid renaming for that purpose, since there are no program variables in ρH and $\rho\pi$. Moreover, there is an i such that any variable $V \in (vars\ H) \cup (vars\ \pi)$ belongs to Var_j for some $j \leq i$. Thus there are always infinitely many unused variables available, for further renaming. We write $V^\#$ for ρV, and $H^\#$ and $\pi^\#$ for the extension of ρ to H and π. The purpose of using ρ for renaming is merely to simplify the definition of one of the primitive operators (the operator $match$), and the model CLP($\mathcal{X}$)$_{pat}$ as a whole does not rely on the choice of using a fixed renaming.

2.3 Informal operational semantics

A CLP($\mathcal{X}$) program is evaluated in a top-down manner as in Prolog. This is defined in terms of query rewriting, which relates answer constraints to initial queries. A *query* $\langle G, \pi \rangle$ to a program P consists of a sequence G of atomic formulas and a constraint π. A *derivation step* relates two queries. Q' is obtained from Q in a derivation step if the following hold:

When Q has the form $\langle \pi :: G, \pi_{acc} \rangle$:

- the constraint $\pi \wedge \pi_{acc}$ is satisfiable;
- and Q' is the query $\langle G, \pi \wedge \pi_{acc} \rangle$.

Or, when Q has the form $\langle A :: G, \pi_{acc} \rangle$:

- there exists a clause C in P, such that clause ($H' :- B'$) is obtained by renaming clause C away from query $\langle A :: G, \pi_{acc} \rangle$
- the constraint $(A = H') \wedge \pi_{acc}$ is satisfiable;
- and Q' is the query $\langle B' :: G, (A = H') \wedge \pi_{acc} \rangle$.

In an *initial query* $\langle A, \pi_{in} \rangle$, the goal consists of a single atom A. A constraint π_{out} is an *answer constraint* associated to a given initial query if the query $\langle nil, \pi_{out} \rangle$ can be obtained from the initial query in a sequence of derivation steps. nil is the empty sequence of atomic formulae.

An interpreter for CLP($\mathcal{X}$) searches for answer constraints to the initial query, using the clauses of the given program. We have assumed a fixed computation rule where the leftmost atom in a query's goal is always chosen to resolve with, but other than that, the way this search is conducted is left open in our definition of CLP($\mathcal{X}$). An interpreter may encounter infinite sequences of derivation steps as well as sequences leading to queries that cannot be rewritten (although the goal is non-empty); since these sequences do not produce answer constraints, they are not reflected in the top-down semantics.

2.4 Lattice theory and fixpoints

Recall that $F : D \rightarrow E$ is monotonic if $d \sqsubseteq d' \Rightarrow Fd \sqsubseteq Fd'$ holds for any d, d', and continuous if $\sqcup\{F\,d \mid d \in C\} = F\,(\sqcup C)$ holds for any chain $C \subseteq D$. When $\sqcup D$ and $\sqcup E$ exist we say F is co-strict if $F(\sqcup D) = \sqcup E$. The sets of syntactic objects $Atom$, Con, $Clause$, and $Progr$ are assumed to be ordered by the identity relation. The compound lattices $D \times E$, D^* (finite sequences of elements from D), and $D \rightarrow E$ are assumed to be ordered "pointwise". The power set $\wp D$ is assumed to be ordered by set inclusion.

Let F be a function on the complete lattice D. For any $n \in \mathcal{N}$ (the natural numbers) we let $F\!\uparrow\! n$ denote the element of D defined inductively by:

$$\begin{array}{lcl} F\!\uparrow\! 0 & = & \sqcap D \\ F\!\uparrow\! n & = & F\,(F\!\uparrow\!(n-1)) \quad (n \geq 1) \end{array}$$

We use that if $F : D \rightarrow D$ is continuous, then the least fixpoint $fix\ F$ of F exist and is equal to $\sqcup\{F\!\uparrow\! n \mid n \in \mathcal{N}\}$; and if F is monotonic, and D is ascending chain finite, then the least fixpoint is equal to $F\!\uparrow\! n$ for some n.

3 Standard interpretation in CLP($\mathcal{X}$)$_{pat}$

This section defines the semantic function $\mathbf{P}^\sharp$, explains the definition of the underlying primitive operators, and establishes the equivalence of the pattern-oriented semantics $\mathbf{P}^\sharp$ with a definition given by Marriott and Søndergaard.

3.1 The standard top-down semantics $\mathbf{P}^\sharp$

The function is intended to capture the standard (or concrete) computation of answer constraints in response to an initial query to a program, *i.e.* as laid down in the sketch of a top-down evaluation model in Section 2. Input and output of clauses is in the form of sets of constrained atoms, a constrained atom being a pair $\langle A, \pi\rangle \in Atom \times Con$. Because of this format, the semantics can form the basis for dataflow analyses that operate on domains of call and success patterns.

It should be noted that information about answer constraints (the *result* of evaluating some initial query) is not what is usually of interest in an application of dataflow analysis. In most cases one is interested in the way that predicates are called (*during* evaluation of a query). This may be the basis for optimizing the code for unification, or the code for solving constraints in a clause body. To capture these aspects of program behavior one must extend the definition of the semantics $\mathbf{P}^\sharp$ accordingly. This complicates the definition somewhat, and since the issues discussed here concerning the representation of input and output of clauses remain the same, it seems justified to work with the simpler semantics $\mathbf{P}^\sharp$.

The definition of $\mathbf{P}^\sharp$ is in terms of a collection of operators which are defined in Figure 4. The function $exists : \wp\, Var \to \wp\, Con \to \wp\, Con$ is defined by $exists\ W\ \Pi = \{\exists_{\overline{W}}.\pi \mid \pi \in \Pi\}$.

Definition 1 The top-down semantics $\mathbf{P}^\sharp$ is defined as follows (with $\phi \in S^\sharp = \wp\,(Atom \times Con) \to \wp\,(Atom \times Con)$).

$$
\begin{array}{lcl}
\mathbf{P}^\sharp\ [\![P]\!] & = & fix\ (\bigcup_{C\in P}\ (\mathbf{C}^\sharp[\![C]\!])) \\
\mathbf{C}^\sharp\ [\![H :- B]\!]\ \phi\ \Delta & = & \text{let } \Pi = match\ H\ \Delta \text{ in} \\
 & & \quad \text{let } \Pi' = \mathbf{B}^\sharp\ [\![B]\!]\ \phi\ \Pi \text{ in} \\
 & & \quad\quad project\ H\ \Pi' \\
\mathbf{B}^\sharp\ [\![nil]\!]\ \phi\ \Pi & = & \Pi \\
\mathbf{B}^\sharp\ [\![A :: B]\!]\ \phi\ \Pi & = & \mathbf{B}^\sharp\ [\![B]\!]\ \phi\ (\mathbf{A}^\sharp\ [\![A]\!]\ \phi\ \Pi) \\
\mathbf{A}^\sharp\ [\![A]\!]\ \phi\ \Pi & = & \text{let } \Delta_{call} = project\ A\ \{\pi\} \text{ in} \\
 & & \quad \text{let } \Delta_{succ} = \phi\ \Delta_{call} \text{ in} \\
 & & \quad\quad \text{let } \Pi' = match\ A\ \Delta_{succ} \text{ in} \\
 & & \quad\quad\quad \bigcup_{\pi\in\Pi}\ (add\ \pi\ \Pi')
\end{array}
$$

□

For any program P, we have that $\mathbf{P}^\sharp\ [\![P]\!]$ is well-defined, because $S^\sharp$ is complete and $\bigcup_{C\in P}\ (\mathbf{C}^\sharp[\![C]\!])$ is monotonic (in fact, continuous).

In Definition 1 it is assumed, for simplicity, that clause bodies contain only atoms. An extension of the definition so as to model the evaluation of constraints can be accomplished by adding an extra rule for $\mathbf{A}^\sharp$:

$$\mathbf{A}^\sharp\ [\![\pi]\!]\ \phi\ \Pi = add\ \pi\ \Pi$$

Analogously, in the development of *analyses* of CLP languages, the definition of how constraints in clause bodies are evaluated may be supplied in a later step. This holds in spite of the fact that the ability to handle constraints is what gives extra expressive power to CLP languages as compared to Prolog; and reflects that the major challenge faced by a CLP or Prolog analysis framework is the modelling of input and output to and from clauses is, not the modelling of the evaluation of constrains in clause bodies.

$$
\begin{array}{lll}
meet & : & \wp\, Con \to \wp\, Con \to \wp\, Con \\
add & : & Con \to \wp\, Con \to \wp\, Con \\
project & : & Atom \to \wp\, Con \to \wp\, (Atom \times Con) \\
match & : & Atom \to \wp\, (Atom \times Con) \to \wp\, Con \\
 & & \\
meet\ \Pi\, \Pi' & = & \{\pi \wedge \pi' \mid \pi \in \Pi \wedge \pi' \in \Pi' \wedge sat(\pi \wedge \pi')\} \\
add\ \pi\, \Pi & = & meet\ \{\pi\}\, \Pi \\
project\ A\, \Pi & = & \{\langle A, \pi\rangle \mid \pi \in \Pi\} \\
match\ H\, \Delta & = & exists\,(vars\ H)\,\{(H = A^{\#}) \wedge \pi^{\#} \mid \langle A, \pi\rangle \in \Delta \wedge sat((H = A^{\#}) \wedge \pi^{\#})\}
\end{array}
$$

Figure 4: The primitive operators.

3.2 The primitive operators

Clause input/output in the format of sets of constrained atoms is handled by the operators $project$ and $match$. The data flowing inside a clause is in the form of a set of constraints; an application of $meet$ conjoins two such sets. Consider the Prolog program of Figure 5 for naive list reversal. We assume the standard Prolog notation for lists, and let $\phi_{fix} = \mathbf{P}^{\sharp}\, [\![P_{rev}]\!]$. We exemplify the use of the three operators by considering the evaluation of a call $rev([2, 1], L)$ in the context of constraint $true$ against the recursive clause C for rev. This is the evaluation of

$$\mathbf{C}^{\sharp}\, [\![C]\!]\, \phi_{fix}\, (project\ rev([2, 1], L)\, \{true\})$$

$project$: The operator is applied to produce input in the form of a set of constrained atoms. (In this case the set is a singleton).

$$project\ rev([2, 1], L)\, \{true\} \quad = \quad \{\langle rev([2, 1], L), true\rangle\}$$

$match$: The application of the operator represents the process of unifying calling atom $rev([2, 1], L)$ and clause head $rev([H|T], Rev)$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{match\ rev([H|T], Rev)\, \{\langle rev([2, 1], L), true\rangle\}} \\
 = & \{\exists L^{\#}.rev([H|T], Rev) = rev([2, 1], L^{\#})\} \\
 = & \{H = 2 \wedge T = [1]\}
\end{array}
$$

The operator $match$, as applied above, can be viewed as proceeding in three steps:

- The constrained atom $\langle rev([2, 1], L), true\rangle$ is renamed to $\langle rev([2, 1], L^{\#}), true\rangle$.
- An equation is set up between the (renamed) calling atom and the clause head.

```
rev([], []).
rev([H|T], Rev) : −rev(T, RevT), app(RevT, [H], Rev).
app([], L, L).
app([H|L1], L2, [H|L3) : −app(L1, L2, L3).
```

Figure 5: P_{rev}: a Prolog program for (naive) list reversal.

- The resulting constraint is quantified existentially with respect to variable $L^{\#}$, producing a constraint solely in terms of the variables in the clause.

Thus, evaluation of the clause body for C begins with (the set consiting of) the constraint $H = 2 \wedge T = [1]$. Note that the two free variables in this constraint occur in the clause itself. In general, the application of $match$ to the constrained atoms supplied as input to a clause implies that any free variable V occuring in the constraints accumulated during the evaluation of a clause, must occur in the clause itself.

$meet$: An application of the operator $meet$ conjoins the information accumulated in the course of evaluating the clause body, with the information arising from evaluating a call to an atom in the clause body. For instance, evaluating the call to $app(RevT, [H], Rev)$ produces the constraint $\pi_{app} : RevT = [1] \wedge H = 2 \wedge Rev = [1, 2]$, which is to be conjoined with the accumulated constraint: $\pi_{acc} : H = [2] \wedge T = [1] \wedge RevT = [1]$:

$$\begin{aligned} & meet\ \{\pi_{app}\}\ \{\pi_{acc}\} \\ & \quad = \ \{RevT = [1] \wedge H = 2 \wedge Rev = [1, 2] \wedge T = [1]\} \end{aligned}$$

Finally, yet an application of $project$ produces the result of evaluating clause C:

$$\begin{aligned} & \mathbf{C}^{\sharp}\ [\![C]\!]\ \phi_{fix}\ \{\langle rev([2, 1], L), true\rangle\} \\ & \quad = \ project\ rev([H|T], Rev)\ \{RevT = [1] \wedge H = 2 \wedge Rev = [1, 2] \wedge T = [1]\} \\ & \quad = \ \{\langle rev([H|T], Rev), RevT = [1] \wedge H = 2 \wedge Rev = [1, 2] \wedge T = [1]\rangle\} \end{aligned}$$

In general, input to a clause is a set of constrainted atoms $\langle A, \pi\rangle$ where A is a calling atom, and output is a set of constrained atoms $\langle H, \pi'\rangle$, where H is the clause head. The result of evaluating different clauses defining the same predicate is the union of the sets of constrained atoms associated with each clause.

3.3 Correctness via Marriott and Søndergaard's semantics

The semantics $\mathbf{P}^{\sharp}$ is intended to capture the top-down evaluation of an arbitrary CLP($\mathcal{X}$) program P, cf. the notion of answer constraints defined informally in

Section 2. We may express this in terms of the semantic function $\mathbf{A}^\sharp$. For instance, the above computation for $rev([2,1],L)$ can be expressed as follows in terms of $\mathbf{A}^\sharp$, indicating that the function in a sense represents the constraint-oriented approach:

$$\mathbf{A}^\sharp \, [\![rev([2,1],L)]\!] \, \phi_{fix} \, \{true\} \quad = \quad \{L = [1,2]\}$$

It is the intention that the semantics $\mathbf{P}^\sharp$ is *correct* with respect to the evaluation model of Section 2 in the following sense: Consider an arbitrary initial query $\langle A, \pi \rangle$ to program P, and assume $(vars\ \pi) \subseteq (vars\ A)$. Then π_{ans} is an answer constraint to the query if and only if

$$exists\ (vars\ A)\ \pi_{ans} = \mathbf{A}^\sharp \, [\![A]\!] \, (\mathbf{P}^\sharp \, [\![P]\!]) \, \{\pi\}$$

We establish the equivalence of $\mathbf{P}^\sharp$ with the semantics $\mathbf{P}_{van}$ defined below. The subscript van refers to the so-called Vanilla interpreter for Prolog (see Sterling and Shapiro [23]), because the semantics can be viewed as evaluating clauses in a straightforward, recursive-descent manner. The semantics $\mathbf{P}_{van}$ is of particular interest because it is, for the case of Prolog, identical to a semantics proposed in Marriott *et al.* [20]. (In [20] the semantics is referred to as the base semantics). In the Vanilla semantics, unification is represented by the operator *unify* : $Atom \rightarrow Atom \rightarrow \wp\, Con \rightarrow \wp\, Con$ defined by

$$\textit{unify}\ H\ A\ \Pi = \bigcup_{\pi \in \Pi} (exists\ (vars\ H)\ (add\ (H = A^\#) \wedge \pi^\#)$$

The function is identical to the function used in [20] to represent unification. This is in the sense that the two functions produce equivalent constraints (for Prolog, equational constraints). Specifically, the constraints differ only in the names of those variables that are quantified existentially.

Definition 2 The Vanilla semantics $\mathbf{P}_{van}$ is defined as follows (with $\phi \in Atom \rightarrow \wp\, Con \rightarrow \wp\, Con$).

$$\begin{array}{lcl}
\mathbf{P}_{van}\, [\![P]\!] & = & fix\ (\bigcup_{C \in P} (\mathbf{C}_{van} [\![C]\!])) \\
\mathbf{C}_{van}\, [\![H :- B]\!]\, \phi\ A\ \Pi & = & \text{let } \Pi' = \textit{unify}\ H\ A\ \pi \text{ in} \\
 & & \quad \text{let } \Pi'' = \mathbf{B}_{van}\, [\![B]\!]\, \phi\ \Pi' \text{ in} \\
 & & \quad \displaystyle\bigcup_{\pi \in \Pi} (add\ \pi\ (\textit{unify}\ A\ H\ \Pi'')) \\
\mathbf{B}_{van}\, [\![nil]\!]\, \phi\ \Pi & = & \Pi \\
\mathbf{B}_{van}\, [\![A :: B]\!]\, \phi\ \Pi & = & \mathbf{B}_{van}\, [\![B]\!]\, \phi\ (\mathbf{A}_{van}\, [\![A]\!]\, \phi\ \Pi) \\
\mathbf{A}_{van}\, [\![A]\!]\, \phi\ \Pi & = & \phi\ A\ \Pi
\end{array}$$

□

The equivalence between $\mathbf{P}_{van}$ and $\mathbf{P}^\sharp$ is established using the following lemma, which follows immediately from the definition of $match$ and *unify*.

Lemma 3 *unify* $H\ A\ \Pi = match\ H\ (project\ A\ \Pi)$.

Proposition 4 $\mathbf{A}^\sharp\ [\![A]\!]\ (\mathbf{P}^\sharp\ [\![P]\!])\ \Pi = \mathbf{P}_{van}\ [\![P]\!]\ A\ \Pi$

The proposition is proved in Appendix A.

The equivalence with Marriott and Søndergaard's constraint-oriented semantic definition for Prolog emphasizes that by the pattern-oriented approach, we cannot expect to attain better analysis precision; rather the issue is whether the pattern-approach may sometimes lead to analyses that are more natural and intuitive, or are better suited to justify the elimination of certain redundant work in the computation of a program's abstract semantics.

4 Non-standard interpretation in CLP$(\mathcal{X})_{pat}$

A dataflow analysis based on call and success patterns can be obtained by providing definitions of abstract domains and operators within the format of *analysis instances* which is set up in this section. For analysis instance X with abstract operators that approximate the standard ones, Proposition 8 (see below) guarantees that the semantic function $\mathbf{P}^\sharp_X$ defined in terms of the abstract operators, approximates the semantics $\mathbf{P}^\sharp$. This means that for any program P, the function $\mathbf{P}^\sharp_X\ [\![P]\!]$ provides correct information about the runtime behavior of the program. For brevity of the presentation, we define the notion of approximation only with respect to the semantic function $\mathbf{P}^\sharp$ and the primitive operators. However, the notion as defined here is consistent with the more general notion of approximation defined in [20].

An analysis instance comprises two abstract domains. Elements of the abstract domain Con_X are used to describe sets of constraints, while elements of Pat_X are used to describe sets of constrained atoms. The elements of Pat_X are the call and success patterns, and the inclusion of this domain is what distinguishes the model presented here from most other frameworks for Prolog or CLP analysis.

The constraint part of an analysis instance is constituted by a so-called constraint analysis instance. This part of a dataflow analysis may be defined in a first phase, cf. the definition in Section 5 of two domains for describing sets of constraints by propositional formulas, prior to the definition of two complete analyses in Sections 6–7.

Definition 5 A *constraint analysis instance* $X = \{Con_X, \gamma^c_X, meet_X, add_X\}$ consists of the following:

- A complete lattice Con_X.
- A concretization function $\gamma^c_X : Con_X \to \wp\, Con$ required to be monotonic, injective, and co-strict.
- Two abstract operators

$$\begin{array}{lcl} meet_X & : & Con_X \to Con_X \to Con_X \\ add_X & : & Con \to Con_X \to Con_X \end{array}$$

$\square$

The elements of Con_X are sometimes referred to as abstract constraints. The abstract operators $meet_X$ and add_X are said to *approximate* the standard operator $meet$ if the following inclusion holds for any pair of abstract constraints $\Pi, \Pi' \in Con_X$ and any $\pi \in Con$:

$$\begin{array}{ll} meet\,(\gamma^c_X\,\Pi)\,(\gamma^c_X\,\Pi') & \subseteq \gamma^c_X\,(meet_X\,\Pi\,\Pi') \\ add\,\pi\,(\gamma^c_X\,\Pi) & \subseteq \gamma^c_X\,(add_X\,\pi\,\Pi) \end{array}$$

Definition 6
An *analysis instance* $X = \{Con_X, Pat_X, \gamma^c_X, \gamma^p_X, meet_X, add_X, project_X, match_X\}$ consists of constraint analysis instance $\{Con_X, \gamma^c_X, meet_X, add_X\}$ in conjunction with the following:

- A complete lattice Pat_X.
- A concretization function $\gamma^p_X : Pat_X \to \wp\,(Atom \times Con)$ required to be monotonic, injective, and co-strict.
- Two abstract operators:

$$\begin{array}{lcl} project_X & : & Atom \to Con_X \to Pat_X \\ match_X & : & Atom \to Pat_X \to Con_X \end{array}$$

$\square$

The elements of Pat_X are usually referred to as call and success patterns. Approximation of the standard operators $project$ and $match$ is defined in analogy with the definition for $meet$, by the requirement that the following inclusions hold, for any pattern Δ, abstract constraint Π and atom $A \in Atom$:

$$\begin{array}{ll} project\,A\,(\gamma^c_X\,\Pi) & \subseteq \gamma^p_X\,(project_X\,A\,\Pi) \\ match\,A\,(\gamma^p_X\,\Delta) & \subseteq \gamma^c_X\,(match_X\,A\,\Delta) \end{array}$$

Definition 7 For a given analysis instance X, the (top-down) analysis semantics $\mathbf{P}^\sharp_X$ is defined as follows (with $\phi \in S^\sharp_X = Pat_X \rightarrow Pat_X$).

$$
\begin{array}{lcl}
\mathbf{P}^\sharp_X \; [\![P]\!] & = & fix\,(\bigsqcup_{C\in P}\,(\mathbf{C}^\sharp_X[\![C]\!])) \\
\mathbf{C}^\sharp_X \; [\![H :- B]\!]\;\phi\;\Delta & = & \text{let } \Pi = match_X\; H\;\Delta \text{ in} \\
 & & \quad \text{let } \Pi' = \mathbf{B}^\sharp_X\;[\![B]\!]\;\phi\;\Pi \text{ in} \\
 & & \qquad project_X\; H\;\Pi' \\
\mathbf{B}^\sharp_X \; [\![nil]\!]\;\phi\;\Pi & = & \Pi \\
\mathbf{B}^\sharp_X \; [\![A :: B]\!]\;\phi\;\Pi & = & \mathbf{B}^\sharp_X\;[\![B]\!]\;\phi\;(\mathbf{A}^\sharp_X\;[\![A]\!]\;\phi\;\Pi) \\
\mathbf{A}^\sharp_X \; [\![A]\!]\;\phi\;\Pi & = & \text{let } \Delta_{call} = project_X\; A\;\Pi \text{ in} \\
 & & \quad \text{let } \Delta_{succ} = \phi\;\Delta_{call} \text{ in} \\
 & & \qquad \text{let } \Pi' = match_X\; A\;\Delta_{succ} \text{ in} \\
 & & \qquad\quad meet_X\;\Pi\;\Pi'
\end{array}
$$

□

The function P_X is well-defined for any analysis instance X. This is due to monotonicity of $\bigsqcup_{C\in P}\,(C_X[\![C]\!])$ and completeness of $S^\sharp_X$, which guarantees the existence of the least fixpoint.

We can define an analysis instance *std* in terms of the standard operators and domains (and with the concretization functions being identity functions). The functions $\mathbf{P}^\sharp_{std}$ and $\mathbf{P}^\sharp$ have identical definitions, except that $\mathbf{A}^\sharp$ which is defined in terms of an expression of the form

$$\bigcup_{\pi\in\Pi}\,(add\;\pi\;(F\;\{\pi\}))$$

is replaced by $\mathbf{A}^\sharp_{std}$ which is defined in terms of an expression in the form

$$meet\;\Pi\;(F\;\Pi)$$

The motivation for this replacement is that in the abstract semantics, an abstract constraint — as used to describe sets of constraints — is non-decomposable. Thus we cannot represent abstractly the operation of selecting individual elements of a set of constraints. The replacement is safe in the sense that $\mathbf{P}^\sharp\,[\![P]\!]\Delta \subseteq \mathbf{P}^\sharp_{std}\,[\![P]\!]\Delta$ holds. This can be verified using that for any monotonic F we have

$$
\begin{array}{lcl}
\bigcup_{\pi\in\Pi}\,(add\;\pi\;(F\;\{\pi\})) & \subseteq & \bigcup_{\pi\in\Pi}\,(add\;\pi\;(F\;\Pi)) \\
 & \subseteq & meet\;\Pi\;(F\;\Pi)
\end{array}
$$

For analysis instance X, the analysis semantics $\mathbf{P}^\sharp_X$ is said to approximate the semantics $\mathbf{P}^\sharp$ if the following inclusion holds:

$$\mathbf{P}^\sharp \, [\![P]\!] \, (\gamma^p_X \, \Delta) \subseteq \gamma^p_X \, (\mathbf{P}^\sharp_X \, [\![P]\!] \, \Delta)$$

This can be read as: "If an input set of constrained atoms to program P is described by Δ, the associated ouput set of constrained atoms is described by $\mathbf{P}^\sharp_X \, [\![P]\!] \, \Delta$". The correctness of the following "approximation proposition" can be established quite straightforwardly using the theory in [20].

Proposition 8 *If the abstract operators of analysis instance X approximate the standard operators, the analysis semantics $\mathbf{P}^\sharp_X$ approximates $\mathbf{P}^\sharp$.*

5 Groundness analysis of Prolog using propositional formulas

This section defines two *constraint analysis instances*, referred to as *conj* and *pos*. In subsequent sections, two complete analyses instances are defined, each on the basis of one of the propositional domains defined in the present section. The analyses are defined for Prolog, rather than for CLP($\mathcal{X}$) in general, because the definition of correct and precise abstract operators is dependent upon the semantics of constraints.

The idea of using propositional formulas for groundness analysis is usually credited to Marriott and Søndergaard's [19]. The subject is already well understood — the material presented here is based on [18] and Cortesi *et al.* [4]. It is recapitulated so that the two pattern-based Prolog analyses can be defined rigorously, as a basis for comparing them with constraint-oriented analyses.

The basic idea is to identify each program variable with a proposition having the same name. A constraint grounding program variable V is described by propositional formula $\mathbf{\Psi}$ if proposition V is true in all models of $\mathbf{\Psi}$. For instance, constraint $H = 2 \wedge T = [1]$ which grounds both variables H and T is described by the conjunction $H \wedge T$, because in any model of formula $H \wedge T$ both propositions H and T must be true.

5.1 Preliminaries

Semantics of constraints in Prolog: We first define the semantics of the constraints that are generated during evaluation of a Prolog program. Con consists of equations between atoms, and the conjunction and existential quantification of such constraints. We need to define $sat \subseteq Con$ so that '=' is interpreted as syntactic identity, in order to obtain Prolog as a CLP($\mathcal{X}$) language.

A substitution is a function $\sigma \in Var \rightarrow Term$ where $\sigma V = V$ for all but finitely many variables. The domain $dom\sigma$ of substitution σ is the set $\{V \mid \sigma V \neq V\}$. A substitution is identified with it's natural extension to atoms. Substitution σ is a solution to constraint $A = H$ if σA and σH are syntactically identical and contain no variables. σ is a solution to constraint $\pi \wedge \pi'$ if it is a solution to both constraints. σ is a solution to constraint $\exists V.\pi$ if there exists a solution σ' to π such that σ' is the same as σ except that possibly $\sigma' V \neq \sigma V$. Finally, $sat \subseteq Con$ is defined by: $sat(\pi)$ holds if π has a solution. In the sequel we also employ the notion of most general unifiers (*mgu*'s), as defined in the usual way (see Lloyd [13]).

Groundness: The function $grounds : Con \rightarrow \wp\, Var$ is defined by:

$$V \in (grounds\ \pi) \Leftrightarrow card\{\sigma V \mid \sigma \text{ is a solution to } \pi\} \leq 1$$

(We write $card\ S$ for the cardinality of set S). If V belongs to the set $grounds\ \pi$ we say that π grounds V. By the above definition, this means that all solutions to π map V to the same value. For instance,

$$\begin{array}{lcl} (grounds\ (H = 2)) & = & \{H\} \\ (grounds\ (L = [H|T])) & = & \emptyset \\ (grounds\ \mathit{false}) & = & Var \end{array}$$

The following lemma can be shown to hold. It is used to establish (in Appendices B–C) that the abstract variants of $match$ approximate the standard variant, which performs variable renaming (see Figure 4 for the standard definition of $match$). (Recall that we write $\pi^{\#}$ for $\rho\pi$).

Lemma 9 $V \in (grounds\ \pi)$ *if and only if* $V^{\#} \in (grounds\ \pi^{\#})$.

The propositional calculus: Let $\Omega(S, PV)$ be the language of propositional formulas build from a set S of logical connectives, a set PV of propositional variables, and the propositional constant *true*. The connectives $\{\wedge, \neg\}$ suffice to define $\{\vee, \leftrightarrow, \rightarrow\}$ as well, so the language $\Omega(\{\vee, \neg\}, PV)$ defined by

$$\begin{array}{rcll} \Psi \in \ \Omega(\{\vee, \neg\}, PV) ::= & & V & (V \in PV) \\ & | & true & \\ & | & \neg\, \Psi & \\ & | & \Psi \wedge \Psi' & \end{array}$$

gives a language for the full propositional calculus over PV.

An interpretation of language $\Omega(S, PV)$ is a function $I \in PV \to \{true, false\}$, and is identified with the set of propositional variables it maps to $true$. An interpretation I is a model of a formula Ψ, $I \models \Psi$, if the formula is true under the interpretation. The notion of truth is assumed to be defined in the standard way, see for instance Bell and Machover's [1]. This yields $\{U, V\} \models U \wedge V$, *etc*. Formulas Ψ and Ψ' are logically equivalent, $\Psi \Leftrightarrow \Psi'$, if they have the same models. Formula Ψ' is a logical consequence of Ψ, $\Psi \Rightarrow \Psi'$, if every model of Ψ is a model of Ψ'.

5.2 Constraint analyses instances *conj* and *pos*

The domains of interest here are defined in terms of languages that are proper subsets of the full propositional calculus. We write $\Omega(S)$ for $\Omega(S, PV)$, assuming that PV is an appropriate, finite superset of the variables occuring in the particular CLP($\mathcal{X}$) program P subjected analysis. If U is the set of variables in P, then the appropriate set is $PV = U \cup U^\#$. We write $L/\Leftrightarrow$ for the quotient structure of language L with respect to the equivalence relation $\Leftrightarrow$. The shorthand *conj* is for *conjunctions*. The name *pos* is for *positive* formulas; formula Ψ is positive if $PV \models \Psi$.

Definition 10 The constraint analysis instance *conj* is defined as follows:

The domain of abstract constraints Con_{conj} is the set

$$\Omega(\{\wedge\})/\Leftrightarrow \cup \{\mathit{false}\}$$

ordered by $\Rightarrow$. The concretization function $\gamma^c_{conj} : Con_{conj} \to \wp\, Con$ is defined by

$$\gamma^c_{conj}\, \Psi \quad = \quad \{\pi \mid (grounds\ \pi) \models \Psi\}$$

Finally, the abstract operator $meet_{conj} : Con_{conj} \to Con_{conj} \to Con_{conj}$ is defined by:

$$meet_{conj}\, \Psi\, \Psi' \quad = \quad \Psi \wedge \Psi'$$

□

It can be shown that the abstract domain Con_{conj} is complete, and that the concretization function γ^c_{con} is monotonic, injective, and co-strict, as required (cf. Definition 5). Monotonicity means that descriptions are ordered according to the size of the sets of constraints they describe. Injectivity means that no two elements of Con_{conj} describe the same set of constraints, and co-strictness means that $\gamma^c_{con}(true) = Con$, so the abstract domain contains a description of the largest set of constraints. Finally, it is straightforward to verify that the abstract operator $meet_{conj}$ approximates the standard operator $meet$. Definition 10 does not specify add_{conj}, and the discussion of *conj* in Section 6 is valid for any (correct) definition of add_{conj}.

In Definition 11 of analysis instance *pos* we make use of the function $eq : Con \to Con_{pos}$ which is defined for constraints that are equations between atoms. Consider the equation

$$\pi : p(a, X) = p(Y, Z)$$

It is clear that by this constraint, Y is ground, and X is ground if and only if Z is ground. Thus $\pi \in \gamma^c_{pos}(Y \wedge (X \leftrightarrow Z))$. The formula $Y \wedge (X \leftrightarrow Z)$ is the value of $eq(\pi)$. In general, the value of $eq(A = H)$ is the strongest formula $\mathbf{\Psi}$ such that $(A = H) \in \gamma^c_{pos}\,\mathbf{\Psi}$. Moreover, the definition of eq constructs the formula explicitly, given a procedure for finding a most general unifer (mgu) for $\{A, H\}$. We write $\bigwedge\{\mathbf{\Psi}_1, \ldots, \mathbf{\Psi}_n\}$ for $\mathbf{\Psi}_1 \wedge \ldots \wedge \mathbf{\Psi}_n$, with $\bigwedge\emptyset = true$.

$$eq(A = H) \;=\; \begin{cases} (\bigwedge\{V \leftrightarrow \bigwedge(vars\ \sigma V) \mid V \in dom\ \sigma\}) & \text{if} \quad sat(A = H) \\ \quad \text{(where } \sigma \text{ is a } mgu \text{ of } \{A, H\}) & \\ \textit{false} & \text{otherwise} \end{cases}$$

Definition 11 The constraint analysis instance *pos* is defined in analogy with *conj* on the basis of $\Omega(\{\wedge, \leftrightarrow, \vee\})$, and with the abstract operator $add_{pos} : Con \to Con_{pos} \to Con_{pos}$ being defined by:

$$add_{pos}\,(A = H)\,\mathbf{\Psi} \;=\; eq(A = H) \wedge \mathbf{\Psi}$$

□

It can be shown that

$$add\,(A = H)\,(\gamma^c_{pos}\mathbf{\Psi}) \subseteq \gamma^c_{pos}(add_{pos}\ eq(A = H)\ \mathbf{\Psi})$$

so add_{pos} approximates add.

By abuse of notation, we usually identify an abstract constraint (a class of logically equivalent formulas) with a formula belonging to the equivalence class. Further, we allow existentially quantified formulas: $\exists V.\mathbf{\Psi}$. For each of the languages, any such formula can be rewritten into a logically equivalent formula in which V does not occur and the quantifier has been eliminated. The operation of "projecting" a formula onto a set W of variables occuring in some atom can be represented by quantifying existentially the variables not in W, and this results in a formula in terms of the variables in W only. To represent this, we use the shorthand $\exists_{\overline{W}}.\mathbf{\Psi}$ analogously as for constraints (see Section 2).

Constraint analysis instance *conj* has considerably less expressive power than *pos*. In fact, on the basis of language $\mathbf{\Omega}(\{\wedge, \leftrightarrow\})$, we may define a constraint analysis instance (we call it *def*, as in [18]), having an intermediate degree of expressive power. Consider the constraint $X = Y$. In *conj* this constraint is described by the formula *true*, while in *def* it is described by $X \leftrightarrow Y$, a stronger formula, and so one that contains more information. In turn, *def* has strictly less expressive power than *pos*: the set of constrains $\{X = a, Y = b\}$ is in *def* described by *true*, while within *pos* the stronger formula $X \vee Y$ is available. Recall that

$$\{X = a, Y = b\} \subseteq \gamma^p_{pos}\,(X \vee Y)$$

because $(grounds\,(X = a)) = X$ and $X \models X \vee Y$, and similarly $(grounds\,(Y = b)) \models X \vee Y$.

6 Patterns for a simple groundness analysis: *conj*

In analysis instance *conj*, sets of constrained atoms are described by simple patterns of the form $p(m_1, \ldots, m_n)$ where any mode m_i is either G or A. For instance, the pattern $qs(\mathsf{G}, \mathsf{A})$ represents all constrained atoms $\langle qs(T_1, T_2), \pi\rangle$ where π grounds all variables occurring in term T_1.

Definition 12 The analysis instance *conj* is defined as follows. The domain of abstract constraints Con_{conj}, the concretization function γ^c_{conj}, and the abstract operator $meet_{conj}$ are as defined in Section 5.

Pat_{conj} is the set

$$Pat_{conj} = \{p(m_1 \ldots m_n) \mid m_i \in \{\mathsf{A}, \mathsf{G}\}\}$$

ordered by the relation $\sqsubseteq$, where $p(m_1 \ldots m_n) \sqsubseteq p'(m'_1 \ldots m'_k)$ holds if $p = p'$, $n = k$, and $(m_i = \mathsf{G} \vee m'_i = \mathsf{A})$ for all i $(1 \leq i \leq n)$.

The concretization function $\gamma^p_{conj} : Pat_{conj} \to \wp\,(Atom \times Con)$ is defined by:

$$\gamma^p_{conj}\; p(m_1 \ldots m_n) \;\;=\;\; \{\langle p(T_1 \ldots T_n), \pi\rangle \mid \forall i : (m_i = \mathsf{G} \Rightarrow vars\; T_i \subseteq (grounds\; \pi))\}$$

Finally the abstract operators are defined by:

$$\begin{array}{rcl} project_{conj}\; p(T_1 \ldots T_k)\; \Psi & = & p(m_1 \ldots m_k) \text{ where } m_i = \begin{cases} \mathsf{G} & \text{if } \Psi \Rightarrow \bigwedge (vars\; T_i) \\ \mathsf{A} & \text{otherwise} \end{cases} \\ match_{conj}\; p(T_1 \ldots T_k)\; p(m_1 \ldots m_k) & & \\ & = & \bigwedge \{\text{if } m_i = \mathsf{A} \text{ then } true \text{ else } \bigwedge (vars\; T_i) \mid 1 \le i \le k\} \end{array}$$

□

(For completeness of Pat_{conj} we need to include artificial bottom and top elements, and extend the definitions of the concretization function and the abstract operators accordingly).

Proposition 13 *The abstract operators of* conj *approximate the standard operators.*

The proposition is proved in Appendix B.

Consider the *quicksort* program of Section 1. Suppose we evaluate the recursive clause for qs against call pattern $qs(\mathsf{G}, \mathsf{A})$. Then

$$match_{conj}\; qs([H|T], Sort)\; qs(\mathsf{G}, \mathsf{A}) = H \wedge T$$

Let Ψ and Ψ' be the abstract constraints holding at the time of the calls to $qs(Sm, SortSm)$ and $qs(Bi, SortBi)$, respectively. That is, let

$$\begin{array}{rcl} \Psi & = & H \wedge T \wedge Sm \wedge Bi \\ \Psi' & = & H \wedge T \wedge Sm \wedge Bi \wedge SortSm \end{array}$$

Then

$$\begin{array}{rcl} project_{conj}\; qs(Sm, SortSm)\; \Psi & = & qs(\mathsf{G}, \mathsf{A}) \\ project_{conj}\; qs(Bi, SortBi)\; \Psi' & = & qs(\mathsf{G}, \mathsf{A}) \end{array}$$

since $\Psi \Rightarrow Sm$ and $\Psi' \Rightarrow Bi$. Thus the two recursive calls to the *quicksort* predicate do indeed give rise to the same call pattern, and so the *conj* analysis as formalized within the model CLP$(\mathcal{X})_{pat}$ attains the goal of providing a formal basis for identifying the two goals.

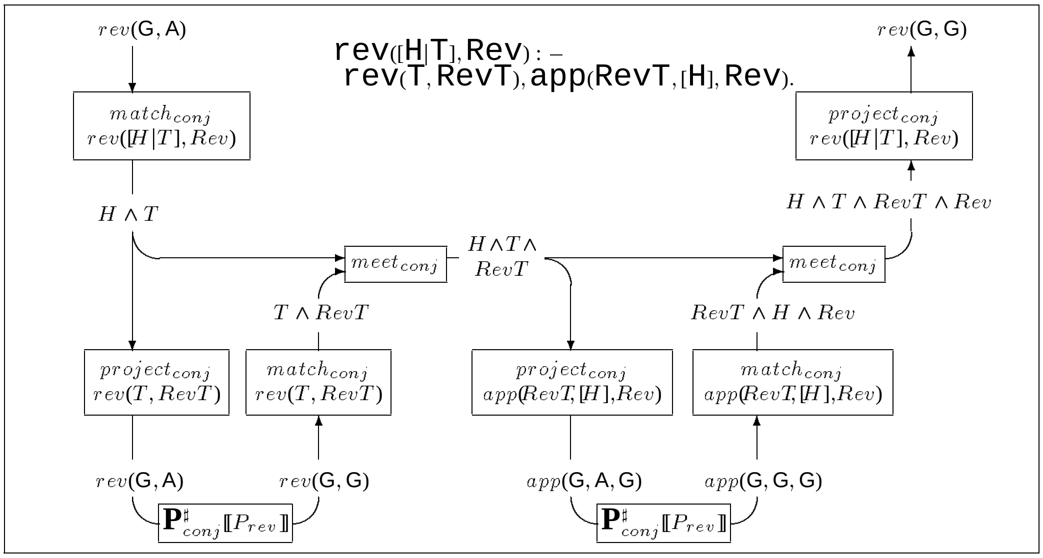

Figure 6: The flow of data through the recursive clause C for rev during groundness analysis using *conj*. Patterns and abstract constraints are indicated on the arrows. Boxes represent functions: either the abstract operators — including their first argument whenever the argument is an atom — or the analysis semantics of the program P_{rev}. The computation indicated is the evaluation of $\mathbf{C}^{\sharp}_{conj}\,[\![C]\!]\,(\mathbf{P}^{\sharp}_{conj}[\![P_{rev}]\!])\,rev(\mathsf{G},\mathsf{A})$, yielding $rev(\mathsf{G},\mathsf{G})$.

As a further example, consider the naive list reversal program of Figure 5. The analysis can infer that if in a call to rev, the first argument is ground, then upon success the second argument is ground as well. Figure 6 gives a graphical illustration of the evaluation of the recursive clause for rev, using that the abstract semantics of the program satisfies

$$\begin{array}{lcl} \mathbf{P}^{\sharp}_{conj}\ [\![P_{rev}]\!]\ rev(\mathsf{G}, \mathsf{A}) & = & rev(\mathsf{G}, \mathsf{G}) \\ \mathbf{P}^{\sharp}_{conj}\ [\![P_{rev}]\!]\ app(\mathsf{G}, \mathsf{G}, \mathsf{A}) & = & app(\mathsf{G}, \mathsf{G}, \mathsf{G}) \end{array}$$

This illustration of the abstract evaluation of a clause for rev provides, in our view, some evidence that analysis instance *conj* provides a fairly understandable and intuitive formulation of a simple Prolog groundness analysis.

In contrast, suppose the *quicksort* program is subjected to a simple groundness analysis defined on the basis of the (constraint-oriented) Vanilla semantics (cf. Section 3). Then in the context of an implementation of the analysis, we may be faced with the problem of justifying that we consider call $qs(Sm, SortSm)$ (with abstract constraint Sm) to be equivalent to call $qs(Bi, SortBi)$ (with abstract constraint Bi), and that we therefore do not re-compute. We may approach this by constructing a template of the form $(qs(Var_1, Var_2), Var_1)$), as proposed in [20]. It is then apparent that both calls are in some sence instances or variants of the template. To formalize the approach, one should lay down how a call is transformed into a template, and vice versa, *etc*. However, this is exactly what is achieved by the pattern-oriented approach, where these considerations are part of the model *per se* (cf. the operators $project$ and $match$, *etc.*). Building these considerations into the model seems to be more reassuring in terms of guaranteeing correctness. Also, defining formally a domain of patterns provides an *interpretation* of patterns, in terms of the standard objects they may be conceived as describing: sets of constrained atoms.

A natural extension of analysis instance *conj* would be to allow patterns such as $p(f(\mathsf{G}, \mathsf{A}), \mathsf{A})$, describing sets of constrained atoms where the first argument is instantiated as indicated. A more comprehensive extension would be to allow information about the freeness of variables (as the basis for this, one would need a more refined constraint analysis instance). It seems that a useful, informal classification of analyses that have a nature similar to that of *conj*, is *attribute independent* analyses, cf. Jones and Muchnick's notion [10]. In the context of constrained atom $\langle A, \pi \rangle$, an attribute independent analysis is one in which the effect of constraint π on variable V occuring in A can only be stated in absolute terms: "V is ground" or "V is free" *etc.*, and not in terms of dependencies with other variables. For these analyses, it is likely that one can define domains of simple patterns having intuitive interpretations. The pattern-oriented approach seems to be useful for this class of relatively simple analyses.

7 Patterns for a very precise groundness analysis: *pos*

The following formalization of an analysis based on positive propositional formulas indicates that the scope of the pattern-oriented approach is not limited to analyses of the crude nature examined in the previous section. We argue that a pattern-based framework is particularly well-suited to formalize a bottom-up (query-independent) analysis based on *pos*. It is well-known that a query-independent variant of the analysis is easier to compute, and provides the same information as the more expensive top-down (or query-dependent) variant. This is due to a property referred to as *condensation* [18], and was confirmed empirically in [24].

The high degree of analysis precision attained when the class of positive formulas is available is due to the ability to express dependency information. For instance, the following true statement about P_{rev} cannot be expressed using *conj*: "Upon success of a call to $rev(T, RevT)$, variable $RevT$ is ground if and only if T is ground". As a consequence we have:

$$\mathbf{P}^{\sharp}_{conj}\ [\![P_{rev}]\!]\ rev(\mathsf{A},\mathsf{G}) = rev(\mathsf{A},\mathsf{G})$$

which means that, unfortunately, the analysis cannot infer that if the second argument is ground at call, then at success the first argument is also ground, although this is actually true. The challenge in the definition of patterns for *pos* is to find a suitable representation of the more precise, dependency-based information.

We first define the analysis semantics $\mathbf{P}^{\flat}_X$ which is a bottom-up counterpart of the top-down analysis semantics $\mathbf{P}^{\sharp}_X$.

Definition 14 For a given analysis instance X, the bottom-up analysis semantics $\mathbf{P}^{\flat}_X$ is defined as follows (with $\Delta \in Pat_X$).

$$\begin{array}{lcl}
\mathbf{P}^{\flat}_X\ [\![P]\!] & = & fix\ (\bigcup_{C \in P}\ (\mathbf{C}^{\flat}_X\ [\![C]\!])) \\
\mathbf{C}^{\flat}_X\ [\![H :- B]\!]\,\Delta & = & \text{let } \Pi = \mathbf{B}^{\flat}_X\ [\![B]\!]\,\Delta\,\{true\} \text{ in } project_X\ H\ \Pi \\
\mathbf{B}^{\flat}_X\ [\![nil]\!]\,\Delta\,\Pi & = & \Pi \\
\mathbf{B}^{\flat}_X\ [\![A :: B]\!]\,\Delta\,\Pi & = & \text{let } \Pi' = \mathbf{A}^{\flat}_X\ [\![A]\!]\,\Delta\,\Pi \text{ in } \mathbf{B}^{\flat}_X\ [\![B]\!]\,\Delta\,\Pi' \\
\mathbf{A}^{\flat}_X\ [\![A]\!]\,\Delta\,\Pi & = & \text{let } \Pi' = match_X\ A\,\Delta \text{ in } meet_X\ \Pi\,\Pi'
\end{array}$$

□

By monotonicity of $\bigcup_{C \in P} (\mathbf{C}^\flat_X [\![C]\!])$ and completeness of Pat_X, the abstract semantics $\mathbf{P}^\flat_X [\![P]\!]$ is well-defined for any P and X.

Note that by Definition 14, the abstract bottom-up semantics of a program is a success pattern (an element of Pat_X). In our bottom-up variant of *pos*, the bottom-up semantics of a program is as follows: Stated informally, the recursive clause for rev is associated with the pair $\langle rev([H|T], Rev), (H \wedge T) \leftrightarrow Rev \rangle$, while the non-recursive clause is associated with $\langle rev([], []), true \rangle$. In general, a clause $H :- B$ is associated with a pair $\langle H, \mathbf{\Psi} \rangle$, where $\mathbf{\Psi}$ is a formula that provides groundness information about the variables in the clause head H, as holding upon evaluation of the clause against an arbitrary call. A success pattern in *pos* is a collection of such pairs of clause heads and formulas.

More formally, the abstract semantics of a program as a whole is modelled as a function $\Delta \in Atom \to Con_{pos}$, where Con_{pos} is the domain of abstract constraints. We identify any pattern Δ with the set of elements $\langle A, \mathbf{\Psi} \rangle \in \Delta$ of the function graph for Δ, where $\mathbf{\Psi}$ is not the bottom element of Con_{pos} (*false*). Thus the abstract semantics Δ associated with a program P contains a pair $\langle H, \mathbf{\Psi} \rangle$ for every clause in the program (assuming all clause heads are distinct). A constrained atom $\langle A, \pi \rangle$, with $\pi \neq$ *false*, is described by pattern Δ if there exists $\langle A, \mathbf{\Psi} \rangle \in \Delta$ such that $\mathbf{\Psi}$ describes π. For instance, constrained atom

$$\{\langle rev([H|T], Rev), H = 2 \wedge T = [1] \rangle\}$$

is described by

$$\{\langle rev([H|T], Rev), H \wedge T \rangle\}$$

We write $\bigvee \{\mathbf{\Psi}_1, \ldots, \mathbf{\Psi}_n\}$ for $\mathbf{\Psi}_1 \vee \ldots \vee \mathbf{\Psi}_n$, with $\bigvee \emptyset =$ *false*.

Definition 15 The analysis instance *pos* is defined as follows, with Con_{pos}, γ^p_{pos}, $meet_{pos}$, and add_{pos} being as defined in Section 5:

$$\begin{array}{lcl}
Pat_{pos} & = & Atom \to Con_{pos} \\
\gamma^p_{pos}\, \mathbf{\Delta} & = & \{\langle A, \pi \rangle \mid \langle A, \mathbf{\Psi} \rangle \in \mathbf{\Delta} \wedge \pi \in \gamma^c_{pos}\, \mathbf{\Psi}\} \\
project_{pos}\, A\, \mathbf{\Psi} & = & \{\langle A, \exists_{\overline{vars\, A}}. \mathbf{\Psi} \rangle\} \\
match_{pos}\, A\, S & = & \exists_{\overline{vars\, A}}. \bigvee \{add_{pos}\, (A = H^\#)\, \mathbf{\Psi}^\# \mid \langle H, \mathbf{\Psi} \rangle \in S\}
\end{array}$$

$\square$

Proposition 16 *The abstract operators of* pos *approximate the standard operators.*

The proposition is proved in Appendix C. The proof is in essence similar to the correctness proof for the analysis defined in [20].

For instance, letting

$$\begin{array}{rcl}
\mathbf{\Psi}_1 & = & add_{pos}\,(app(L1, L2, L3) = app([], L^\#, L^\#))\ true \\
 & = & L1 \wedge (L2 \leftrightarrow L^\#) \wedge (L3 \leftrightarrow L^\#) \\
\mathbf{\Psi}_2 & = & add_{pos}\,(app(L1, L2, L3) = app([H^\#|L1^\#], L2^\#, [H^\#|L3^\#]))\ true \\
 & = & (L1 \leftrightarrow (H^\# \wedge L1^\#)) \wedge (L2 \leftrightarrow L2^\#) \wedge (L3 \leftrightarrow (H^\# \wedge L3^\#))
\end{array}$$

we obtain

$$\begin{array}{l}
match_{pos}\ app(L1, L2, L3)\ \{\langle app([], L, L), true\rangle\} \\
\qquad = \ \exists L^\#.\mathbf{\Psi}_1 \\
\qquad = \ (L1 \wedge L2) \leftrightarrow L3 \\
match_{pos}\ app(L1, L2, L3)\ \{\langle app([H|L1], L2, [H|L3]), (L1 \wedge L2) \leftrightarrow L3\rangle\} \\
\qquad = \ \exists H^\#, L1^\#, L2^\#, L3^\#.\mathbf{\Psi}_2 \ \ \wedge \ \ ((L1^\# \wedge L2^\#) \leftrightarrow L3^\#) \\
\qquad = \ (L1 \wedge L2) \leftrightarrow L3
\end{array}$$

Thus, if Δ is the pattern $\{\langle app([], L, L), true\rangle, \langle app([H|L1], L2, [H|L3]), (L1 \wedge L2) \leftrightarrow L3\rangle\}$, we have

$$match_{pos}\ app(L1, L2, L3)\ \Delta = (L1 \wedge L2) \leftrightarrow L3$$

Consistently with [18] we may define that analysis instance X is condensing if, letting $F_X = \lambda A.(\mathbf{A}^\sharp_X\ [\![A]\!]\ (\mathbf{P}^\sharp_X\ [\![P]\!]))$ for given P, we have for any abstract constraint $\Pi \in Con_X$:

$$F_X\ A\ \Pi = \Pi \wedge (F_X\ A\ \top)$$

The bottom-up semantics $\mathbf{P}^\flat_X$ is useful in the context of a condensing analysis instance X, since in that case

$$F_X\ A\ \Pi = \mathbf{A}^\flat_X\ [\![A]\!](\mathbf{P}^\flat_X\ [\![P]\!])\ \Pi$$

In particular, as it can be shown that *pos* is condensing, a bottom-up variant of *pos* expressed as the abstract semantics $\mathbf{P}^\flat_{pos}$ is useful. For instance, given atom $rev(X, Y)$ and abstract constraint Y, to be evaluated against program P_{rev}, we can find the associated, abstract (answer) constraint as follows:

First, compute $\mathbf{P}^\flat_{pos}\ [\![P_{rev}]\!]$. The result of this is that

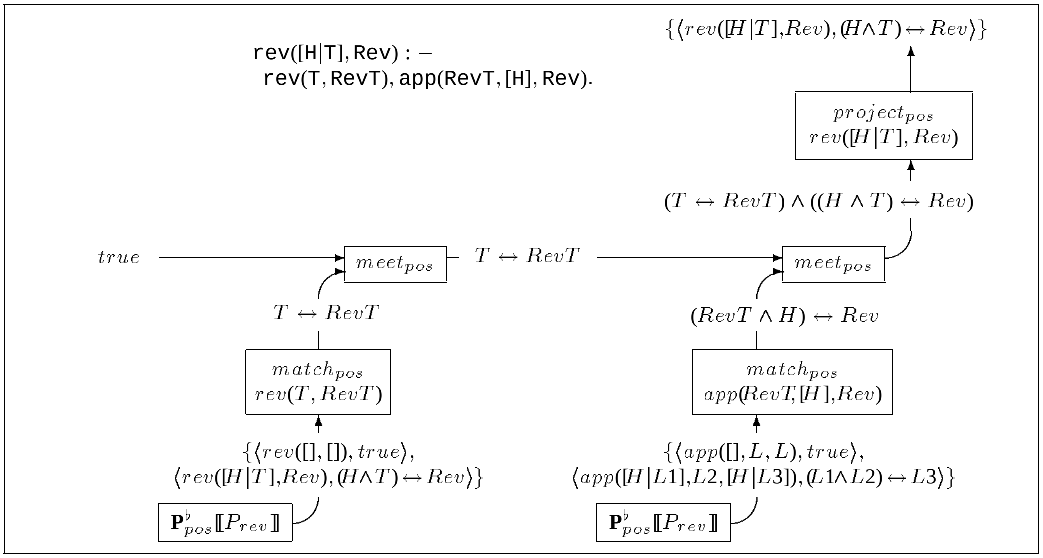

Figure 7: The flow of data through the recursive clause C for rev during groundness analysis using *pos*. Patterns and abstract constraints are indicated on the arrows. Boxes represent functions: either the abstract operators — including their first argument whenever the argument is an atom — or the analysis semantics of the program P_{rev}. The computation indicated is the evaluation of $\mathbf{C}^{\flat}_{pos}\,[\![C]\!]\,(\mathbf{P}^{\flat}_{pos}[\![P_{rev}]\!])$, yielding $\{\langle rev([H|T], Rev), (H \wedge T) \leftrightarrow Rev\rangle\}$.

$$\begin{array}{lcll}\mathbf{P}^{\flat}_{pos}\,[\![P_{rev}]\!] & = & \{ & \langle rev([],[]), true\rangle, \\ & & & \langle rev([H|T], Rev), (H \wedge T) \leftrightarrow Rev\rangle, \\ & & & \langle app([], L, L), true\rangle, \\ & & & \langle app([H|T1], T2, [H|T3]), (T1 \wedge T2) \leftrightarrow T3\rangle \ \}\end{array}$$

(See Figure 7). Second, we obtain

$$\begin{array}{lcl}F_{pos}\,A\,Y & = & \mathbf{A}^{\flat}_{pos}\,[\![rev(X,Y)]\!](\mathbf{P}^{\flat}_{pos}\,[\![P_{app}]\!])\,Y \\ & = & \text{let } \boldsymbol{\Psi} = match_{pos}\,rev(X,Y)\,(\mathbf{P}^{\flat}_{pos}\,[\![P_{app}]\!]) \text{ in } Y \wedge \boldsymbol{\Psi} \\ & = & Y \wedge (X \leftrightarrow Y) \\ & = & X \wedge Y\end{array}$$

Thus we have used *pos* to establish that the grounding of the second argument (Y) implies the grounding of the first (X) as well.

More generally, this computation for rev indicates that in a first step, one may compute information about a program in a query-independent fashion; and then in a second step, one may "query" the result, and in fact obtain the same information as if one had proceeded in a query-dependent manner in the first place. This overall approach is similar to the one sketched in [18].

The above re-formulation of a groundness analysis based on *pos* indicates, first, that a query-independent formulation of the analysis instance *pos* can be expressed naturally using a bottom-up analysis semantics that operates on a domain of patterns. More precisely, the bottom-up analysis operates on *success* patterns. In particular, the abstract semantics $\mathbf{P}^{\flat}_{pos}\,[\![P]\!]$ of a program P as listed above is a success pattern, describing — as usual — a set of constrained atoms.

Second, because of the strategy of working with the original variable names of the program clauses, the bottom-up variant of *pos* assigns to each clause head a propositional formula in terms of the variables of the clause head. This yields an analysis result which is very intuitive.

Alternatively, an analysis based on *pos* can be defined within the context of a top-down semantics as in [18], cf. our Vanilla semantics (Definition 2). Then the representation of the analysis result becomes somewhat arbitrary, because it depends on the choice of variable names in queries. The result for, say rev, might be represented as

$$F_{pos}\,rev(X,Y)\,true = X \leftrightarrow Y$$

but could equally well be represented as

$$F_{pos}\ rev(T, RevT)\ true = T \leftrightarrow RevT$$

It seems preferable to attain a formulation of the analysis in which this arbitrary choice of variable names is eliminated. This is achieved by representing the abstract semantics of a program as a success pattern.

The pattern-based analysis using positive propositional formulas comes at no extra cost in terms of added complexity of correctness proofs for the abstract operators. This is true in spite of the fact that the analysis employs operators $meet_{pos}$, add_{pos}, $project_{pos}$, and $match_{pos}$, whereas a constraint-based analysis can be defined in terms of (only) $meet_{pos}$, add_{pos}, and unify_{pos}, the latter approximating the primitive operator *unify* (as defined in Section 3). The correctness proofs for the two variants of groundness analysis using positive propositional formulas are in substance the same. In particular, we may (consistently with Marriott and Søndergaard's [18]) define

$$\mathit{unify}_{pos}\ A\ H\ \Psi = \exists_{\overline{(vars\ A)}}.(add_{pos}\ (A = H^{\#})\ \Psi^{\#})$$

and then we have (cf. also Lemma 3)

$$\mathit{unify}_{pos}\ A\ H\ \Psi = match_{pos}\ A\ (project_{pos}\ H\ \Psi)$$

This equality emphasizes that while different in flavour, the two approaches are similar in substance, and that placing the issue of disregarding particular variable names *within* the formal framework of CLP($\mathcal{X}$)$_{pat}$ is attained by adding only a minimum of extra formal machinery.

8 Conclusion

This report has proposed the model CLP($\mathcal{X}$)$_{pat}$ in an attempt to answer two questions concerning the abstract interpretation of CLP programs: whether pattern-oriented analyses can be given a rigorous formal basis, and whether such analyses are useful ?

It seems justified to answer the first question in the affirmative. Two well-known groundness analyses, *conj* and *pos*, have been reformulated in terms of call and success patterns within the context of the model. Formal correctness proofs were given for these analyses using a general, and conventional, result that allowed the "lifting" of the property of approximation from a collection of operators to the analysis semantics as a whole (Proposition 8).

The reformulation of the simple analysis *conj* yielded a pattern-based analysis which is very similar to analyses already proposed in the literature. (There are also analyses that incorporate the proposed analysis in a more comprehensive context). The difference to these previous works is that our variant of *conj* comes out as an instance of a generic model. The reformulation of the more precise analysis *pos* yielded an analysis which is in a broad sense similar to previously proposed variants, in having a query-independent or bottom-up flavor, utilizing that *pos* is condensing. Our specific formulation of the *pos* analysis differs from previous proposals in taking the full consequence of this, stating the analysis in terms of a dedicated bottom-up semantics.

The answer "Yes, sometimes" therefore seems admissible in response to the second question. This is on the basis that for the two particular analyses that have been examined, we have found that the pattern-oriented approach does have certain advantages: The *conj* analysis provided a basis for justifying, in a rigorous manner, the elimination of certain redundant computations, for calls that were renamed variants. For both of the two analyses, viewing the input and output of clauses as sets of constrained atoms (cf. the semantics $\mathbf{P}^{\sharp}$) appears to be beneficial. This allows for the definition of domains of patterns that on the one hand possess an unambiguous formal interpretation, namely as descriptions of sets of constrained atoms; and which on the other hand have a pleasant, intuitive reading. We feel this holds (in *conj*) for, say the pattern $qs(\mathsf{G}, \mathsf{A})$ expressing the grounding of the first argument in a call to qs; and (in *pos*) for, say the (fragment of a) pattern $\langle rev([H|T], Rev), (H \wedge T) \leftrightarrow Rev \rangle$, expressing that variable Rev is ground if and only if H and T are ground, in output from a list reversal clause with head $rev([H|T], Rev)$. In view of the fact that dataflow analyses are usually complex and difficult to understand, these small contributions to the formulation of analyses that are not only formally well justified, but also intuitively fairly understandable, seems to validate that call and success patterns have a place in the field of abstract interpretation of CLP languages.

A The equivalence of $\mathbf{P}_{van}$ and $\mathbf{P}^\sharp$

This Appendix contains a proof of Proposition 4 which says that $\mathbf{A}^\sharp[\![A]\!](\mathbf{P}^\sharp[\![P]\!])\Pi = \mathbf{P}_{van}\,[\![P]\!]\,A\,\Pi$.

Proof We consider a given program P. By continuity of $\bigcup_{C\in P}(\mathbf{C}^\sharp[\![C]\!])$ and $\bigcup_{C\in P}(\mathbf{C}_{van}[\![C]\!])$ it suffices to establish that $\forall\, n : s(n)$ holds, where

$$\begin{array}{lcl} s(n) & \Leftrightarrow & \mathbf{A}^\sharp\,[\![A]\!]\,\phi^n\,\Pi = \phi^n_{van}\,A\,\Pi \\ \phi^n & = & (\bigcup_{C\in P}(\mathbf{C}^\sharp[\![C]\!]))\uparrow n \\ \phi^n_{van} & = & (\bigcup_{C\in P}(\mathbf{C}_{van}[\![C]\!]))\uparrow n \end{array}$$

We show this by induction over n.
Base case. $s(0)$ holds trivially, since $\phi^0_{van}\,A\,\Pi = \phi^0\,\Delta = \emptyset$.
Inductive case. We assume $s(n)$ and show $s(n+1)$.
It is straightforward to verify that the equality

$$\mathbf{B}^\sharp\,[\![B]\!]\,\phi^n\,\Pi = \mathbf{B}_{van}\,[\![B]\!]\,\phi^n_{van}\,\Pi \qquad (1)$$

holds, using induction over the length of B. To show $s(n+1)$ it suffices to establish that the following equality holds for an arbitrary clause $H :- B \in P$.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{A}^\sharp\,[\![A]\!]\,(\mathbf{C}^\sharp\,[\![H :- B]\!]\,\phi^n)\,\Pi} \\ = & \text{let } \Pi' = match\ H\ (project\ A\ \Pi) \text{ in} & \\ & \quad \text{let } \Pi'' = \mathbf{B}^\sharp\,[\![B]\!]\,\phi^n\,\Pi' \text{ in} & \\ & \quad \displaystyle\bigcup_{\pi\in\Pi}(add\ \pi\ (match\ A\ (project\ H\ \Pi''))) & \text{(by definition of } \mathbf{A}^\sharp \text{ and } \mathbf{C}^\sharp\text{)} \\ = & \text{let } \Pi' = unify\ H\ A\ \Pi \text{ in} & \\ & \quad \text{let } \Pi'' = \mathbf{B}^\sharp\,[\![B]\!]\,\phi^n\,\Pi' \text{ in} & \\ & \quad \displaystyle\bigcup_{\pi\in\Pi}(add\ \pi\ (unify\ A\ H\ \Pi'')) & \text{(by Lemma 3)} \\ = & \text{let } \Pi' = unify\ H\ A\ \Pi \text{ in} & \\ & \quad \text{let } \Pi'' = \mathbf{B}_{van}\,[\![B]\!]\,\phi^n_{van}\,\Pi' \text{ in} & \\ & \quad \displaystyle\bigcup_{\pi\in\Pi}(add\ \pi\ (unify\ A\ H\ \Pi'')) & \text{(by (1) above)} \\ = & \mathbf{C}_{van}\,[\![H :- B]\!]\,\phi^n_{van}\,\Pi' & \text{(by definition of } \mathbf{C}_{van}\text{)} \end{array}$$

□

B Correctness of *conj*

This appendix gives a proof of Proposition 13 which says that the abstract operators of analysis instance *conj* approximate the standard operators.

Proof

$project_{conj}$: We must show $project\ A\ (\gamma^c_{conj}\ \Psi) \subseteq \gamma^p_{conj}\ (project_{conj}\ A\ \Psi)$. Let $\langle A, \pi\rangle \in project\ A\ (\gamma^c_{conj}\ \Psi)$ with $A = p(A_1 \ldots A_n)$, and let $p(m_1 \ldots m_n) = project_{conj}\ A\ \Psi$. Consider an arbitrary i where $m_i = \mathsf{G}$. By definition of γ^p_{conj} it suffices to establish the inclusion

$$vars\ A_i \subseteq (grounds\ \pi) \tag{2}$$

Since $\pi \in \gamma^c_{conj}\ \Psi$ we have that $(grounds\ \pi) \models \Psi$. By definition of $project_{conj}$ we have that $\Psi \models \bigwedge(vars A_i)$. We infer $(grounds\ \pi) \models \bigwedge(vars A_i)$. Since this means that for any variable $V \in vars\ A_i$, the proposition V is true in the interpretation $(grounds\ \pi)$, the inclusion (2) follows.

$match_{conj}$: We must show $match A(\gamma^p_{conj} p(m_1 \ldots m_n)) \subseteq \gamma^c_{conj}(match_{conj} A p(m_1 \ldots m_n))$. Let $\pi \in match\ A\ (\gamma^p_{conj}\ p(m_1 \ldots m_n))$ with $A = p(A_1 \ldots A_n)$. By definition of γ^c_{conj} we must show that

$$(grounds\ \pi) \models match_{conj}\ A\ p(m_1 \ldots m_n)$$

Consider an arbitrary i where $m_i = \mathsf{G}$. By definition of $match_{conj}$ it suffices to establish:

$$(grounds\ \pi) \models \bigwedge (vars\ A_i) \tag{3}$$

By definition of γ^p_{conj}, we have that $\pi = \exists_{vars\ A}\ (A = H^\#) \wedge \pi'^\#$ for some $\langle H, \pi'\rangle$ where $(vars\ H_i) \subseteq (grounds\ \pi')$ (we let $H = p(H_1 \ldots H_n)$). By Lemma 9 it follows that $(vars\ H^\#_i) \subseteq (grounds\ \pi'^\#)$. Since constraint $\pi'^\#$ determines uniquely the variables in $H^\#_i$, constraint $(A = H^\#) \wedge \pi'^\#$ determines uniquely the variables in A_i as well. The same holds when we quantify existentially over the variables not in A, so $(vars\ A_i) \subseteq (grounds\ \pi)$ holds, from which we infer (3). □

C Correctness of *pos*

This appendix gives a proof of Proposition 16 which says that the abstract operators of analysis instance *pos* approximate the standard operators.

The proof uses that the following can be shown to hold:

$$(grounds\ \pi) \models \mathbf{\Psi}\ \text{ implies }\ (grounds\ \exists V.\pi) \models \exists V.\mathbf{\Psi} \tag{4}$$

Proof

$project_{pos}$: It suffices to note that any constraint described by proposition $\mathbf{\Psi}$ is described by (the weaker) $\exists V.\mathbf{\Psi}$.

$match_{pos}$: We must show $match\,A(\gamma^p_{pos}S) \subseteq \gamma^c_{pos}(match_{pos}AS)$. Let $\langle H, \mathbf{\Psi}\rangle \in S$ and let $\pi' \in match\ A\,(\gamma^p_{pos}\ \{\langle H, \mathbf{\Psi}\rangle\})$. Then by definition of $match$ and γ^p_{pos} we have $\pi' = (\exists_{\overline{vars\,A}}\,(A = H^{\#}) \wedge \pi^{\#})$ where

$$(grounds\ \pi) \models \mathbf{\Psi} \tag{5}$$

By definition of $match_{pos}$ it suffices to show that

$$(grounds\ \exists_{\overline{vars\,A}}\,(A = H^{\#}) \wedge \pi^{\#}) \models \exists_{\overline{vars\,A}}\,(add_{pos}\,(A = H^{\#})\,\mathbf{\Psi}^{\#})$$

By the assumption that add_{pos} approximates add we have that, for all π'' and $\mathbf{\Psi}$

$$add\ \pi''\ \{\pi \mid (grounds\ \pi) \models \mathbf{\Psi}\} \subseteq \{\pi \mid (grounds\ \pi) \models add_{pos}\ \pi''\ \mathbf{\Psi}\} \tag{6}$$

From (5) and Lemma 9 we infer $(grounds\ \pi^{\#}) \models \mathbf{\Psi}^{\#}$, and thus

$$(A = H^{\#} \wedge \pi^{\#}) \in add\,(A = H^{\#})\,\{\pi''' \mid (grounds\ \pi''') \models \mathbf{\Psi}^{\#}\}$$

which by (6) implies

$$(grounds\,(A = H^{\#}) \wedge \pi^{\#}) \models (add_{pos}\,(A = H^{\#})\,\mathbf{\Psi}^{\#})$$

By (4) this completes the proof. □

References

[1] J. Bell and M. Machover. *A course in mathematical logic*. North-Holland,1977.

[2] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming* 1991, **10**, pp. 91–124.

[3] M. Bruynooghe and D. Boulanger. Abstract interpretation for (constraint) logic programming. In: B. Mayoh *et al.* (eds). *Constraint Programming*, NATO Advanced science series, Springer-Verlag, 1994.

[4] A. Cortesi, G. File, and W. Winsborough. *Prop* revisited. Propositional formula as abstract domain for groundness anaysis. *Proc. 6th IEEE Symposium on Logic in Computer Science*, 1991, pp. 322–327.

[5] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proc. 4th ACM Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[6] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, **11** (3), 1989, pp. 418–450.

[7] M. H. Garcia de la Banda, M. Hermenegildo. A practical approach to the global analysis of CLP programs. *Proc. International Logic Programming Symposium*, 1993, pp. 437–455.

[8] T. W. Getzinger. The costs and benefits of abstract interpretation-driven Prolog optimization. Proc. Static Analysis Symposium (SAS) '94, *Lecture Notes in Computer Science 864*, pp. 1 – 25.

[9] J. Jaffar and J.-L. Lassez. *Constraint logic programming.* Technical Report, Monash University, June 1986.

[10] N. Jones and S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra, in N. Jones and S. Muchnick (eds): *Program flow analysis. theory and applications*, Prentice-Hall, 1981, pp. 380–393.

[11] N. Jørgensen. *Abstract interpretation of constraint logic programs.* Ph.D. thesis. Computer Science Dept., Roskilde University, 1992.

[12] T. Kanamori. Abstract interpretation based on Alexander templates. Technical Report 549, ICOT, Tokyo, 1990.

[13] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 1984.

[14] H. Mannila and E. Ukkonen. Flow analysis of Prolog programs. *Proc. 4th IEEE Symposium on Logic Programming*, San Francisco, 1987.

[15] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, pp. 296–306.

[16] B. Le Charlier, K. Musumbu, P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. *Proc. 8th International Conference on Logic Programming*, 1991, pp. 64–78.

[17] K. Marriott and H. Søndergaard. Analysis of constraint logic programs. *Proc. North American Conference on Logic Programming*, Austin, 1998, pp. 521–540.

[18] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on programming languages and systems*, Vol. 2, Nos. 1-4, 1993, pp 181–196.

[19] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. North American Conference on Logic Programming, 1989.

[20] K. Marriott, H. Søndergaard, and N. Jones: Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, May 1994, Vol. 16, No. 3, pp. 607–648.

[21] C. S. Mellish. *The automatic generation of mode declarations for Prolog programs.* DAI Research Paper 163, University of Edinburgh, 1981.

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

[23] L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, 1986.

[24] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The impact of granularity in abstract interpretation of Prolog. *Proc. Workshop on Static Analysis*,1993, LNCS 724, Springer-Verlag, pp. 1-14.

[25] D.H.D. Warren. *Implementing Prolog - compiling predicate logic programs. Vol 1.* DAI Research Paper 39, University of Edinburgh, 1977.

[26] W. Winsborough. Multiple specialization using minimal-function graph semantics. *J. Logic Programming* 1992 **13**, pp. 259–290.