Chaotic fixpoint iteration guided by dynamic dependency

Niels Jørgensen
Roskilde University Center
DK-4000 Roskilde, Denmark
nielsj@dat.ruc.dk

Abstract

An algorithm for abstract interpretation of logic programs is defined and analyzed. The algorithm is proved to be correct with respect to an abstract semantics for (a variant of) Prolog. This abstract semantics associates a given program with a function that maps each call pattern for a predicate to a distinct success pattern. The proposed algorithm employs a variant of chaotic iteration, and is based on what may be termed a dynamic dependency relation. A low worst-case complexity is achieved: the number of passes of dataflow analysis over each program clause is proved to be independent of the size of the rest of the program.

1 Introduction

Compilers for Prolog may produce more efficient low-level code if they employ static analysis to obtain information about a program's runtime behavior. An early proposal for this was made by Mellish [21]. More recent research has looked into the technique's relevance for the compilation of constraint logic programming (CLP) languages, including the case of CLP($\mathcal{R}$) as studied by Jaffar *et al.* [11] and Jørgensen *et al.* [13]. Although some optimizing program transformations may benefit from information about how predicates will succeed, as demonstrated by Marriott and Søndergaard [19], most optimizations rely on information about how predicates will be called during runtime.

Abstract interpretation as originally proposed by Cousot and Cousot [6] provides a mathematical framework for proving an analysis correct with respect to a programming language's operational semantics. Also, a formal framework may serve as a basis for describing and understanding various fixpoint computing strategies, for instance the chaotic iteration strategies discussed in [8]. An abstract interpretation-based approach for the analysis of constraint logic programs is presented by Marriott and Søndergaard [18]. This work suggests that the analysis of Prolog can be given a very simple formulation within a general CLP framework, an approach followed in the present paper.

An algorithm for an analysis problem in the context of imperative languages was proposed by Kildall [17]. This algorithm employs a *static dependency relation* to guide the fixpoint iteration. A similar idea is employed in O'Keefe's algorithm [15], which was shown to apply to the Prolog analysis problem considered by Mellish, and in Debray's algorithm for a more refined variant of mode analysis for Prolog [9].

The top-down analysis considered in the present paper maintains a distinction between different ways in which a predicate is called. The basic analysis problem is similar to the one solved by the algorithm given by Le Charlier *et al.* [4]. The top-down algorithm we propose employs a variant of the idea proposed in [4] of guiding the fixpoint iteration by a *dynamic dependency relation*. That is, the choice of what to compute (which clause and call pattern to evaluate) in an iteration is controlled by a data structure which is modified dynamically during the fixpoint iteration.

The algorithm of [4] has been shown to be optimal for certain classes of programs, as well as to perform very well in practice [5]. The algorithm's absolute worst-case complexity has been analyzed in terms of the total number of times that a clause in the given program is entered for dataflow analysis with respect to a call pattern. This number is shown in [4] to be of the order of $DomSize^2 * DomHeight * Tree^2$, where $DomSize$ is the maximum size of the domain of patterns for a predicate, $DomHeight$ is the height of that domain, and $Tree$ is the size of the so-called static call tree for the given program (that size in turn being of the order of the number $Atoms$ of atoms in the program). The corresponding complexity measure for the top-down algorithm presented in the present paper is of the order $DomSize * DomHeight * Atoms$. Thus, the worst-case bound of [4] is improved by two orders of magnitude.

Our complexity result also implies that the number of times that each clause is subjected to a pass of dataflow analysis is independent of the size of the rest of the program.

Related work includes the frontiers algorithm [22], originally developed for the analysis of functional languages. The algorithm was designed for the computation of a function with a binary codomain (say, $\{\perp, \top\}$). In a logic programming context we would like to operate with a multitude of success patterns, and indeed subsequent research [20] has extended the basic idea in the frontiers algorithm so that an arbitrary finite codomain is allowed.

In a comparative study of a number of techniques for the evaluation of queries to deductive databases, Bancilhon and Ramakrishnan [2] give particular credit to the method of semi-naive or differential evaluation, a generalized variant of which is analyzed in [1]. If used in combination with a program transformation based on magic sets [3], the method may produce information about how predicates are called in a logic program.

The main parts of the paper are sections 4 and 5. Section 4 introduces the notion of a static dependency relation, and defines an algorithm which uses such a relation in the computation of an abstract bottom-up semantics. Section 5 defines a dynamic dependency relation, and employs it in an algorithm for the computation of an abstract top-down semantics. The abstract top-down semantics of a program may express more information about the program than the bottom-up semantics; the relevant fixpoint is more difficult to compute, and emphasis in the presentation is on explaining why and how dynamic dependency is useful for solving the more complex fixpoint problem.

2 Lattices and fixpoints

This section recapitulates some definitions and results from lattice theory.

A complete lattice is a set D equipped with a partial ordering relation $\sqsubseteq$ with the property that for any subset $S \subseteq D$ the least upper bound $\sqcup S$ and the greatest lower bound $\sqcap S$ exist. We write $\perp$ for $\sqcap\ D$ when D is given from the context. If D is such that $\sqcap\ D$ does not exist we sometimes add $\perp$ to D as an "artificial" bottom element. Dually for $\sqcup\ D$ and $\top$. The size of D is the number of elements in D. A subset $S \subseteq D$ is a chain if $d \sqsubseteq d'$ or $d' \sqsubseteq d$ holds for all $d, d' \in S$. The height of D is the maximum size of any chain in D.

Suppose D and E are complete lattices. The Cartesian product $D \times E$, the set D^* of finite sequences of elements of D, and the function space $D \rightarrow E$ (of monotonic and nonmonotonic functions) are all assumed to be ordered pointwise. Under these ordering relations the composite lattices are complete.

The power set of D is written $\mathcal{P}\ D$.

All sets of syntactic objects (programs, clauses, etc.) are assumed to be flat domains, that is, ordered by the identity relation. If D is a flat domain and E is a complete lattice, then the function space $D \rightarrow E$ (ordered pointwise) is a complete lattice; and any function $F : D \rightarrow E$ is monotonic, i.e., $d \sqsubseteq d' \Rightarrow (F\ d) \sqsubseteq (F\ d')$ holds for all $d, d' \in D$.

A fixpoint of a function $F : D \rightarrow D$ is an element $x \in D$ which satisfies $F\ x = x$. Let $S \subseteq D$ be the set of fixpoints of F. If $\sqcap\ S$ exists and is itself a fixpoint of F, then it is called the least fixpoint and denoted $lfp\ F$.

The following theorem gives sufficient criteria for the existence of the least fixpoint of a function. It follows from Tarski's theorem [23]:

Theorem 2.1 *Let D be a complete lattice, and let $F : D \rightarrow D$ be monotonic. Then $lfp\ F$ exists.*

The first of the following two propositions gives sufficient criteria for the existence of an algorithm to compute the least fixpoint of a function. The proposition is expressed using the following notation: Let a function $F : D \rightarrow D$, where D is a complete lattice, be given. Then $F \uparrow n$ denotes the element of D defined inductively by:

$$\begin{aligned} F \uparrow 0 &= \perp \\ F \uparrow n &= F\ (F \uparrow (n-1)) \text{ for } n \in \{1, 2, \ldots\} \end{aligned}$$

Proposition 2.2 *Let D be a complete lattice having finite height, and let $F : D \rightarrow D$ be monotonic. Then there exists $n \in \mathcal{N}$ such that $lfp\ F = F \uparrow n$, and n is at most the height of D.*

The sequence $F \uparrow 0, \ldots, F \uparrow n, \ldots$ is called the ascending Kleene sequence for F.

The second proposition is used in proving the fixpoint algorithms correct. Both parts of the proposition are easily verified.

Proposition 2.3 *Let D be a complete lattice, and let $F : D \rightarrow D$ be monotonic.*
(i) Suppose the sequence $x^0, \ldots, x^n, \ldots$ of elements of D is given by:

$$\begin{array}{lll} x^0 & = & \perp \\ x^n & = & x^{n-1} \sqcup (F_n \ x^{n-1}) \end{array}$$

where $F_n : D \rightarrow D$ and $F_n \preceq F$ holds for all n. Then $x^n \sqsubseteq lfp\ F$ holds for all n.
(ii) Suppose that $x \in D$ is such that $F\ x \sqsubseteq x$. Then $lfp\ F \sqsubseteq x$.

3 An example: groundness analysis of Prolog programs

The example language considered in the paper is a variant of Prolog. The language is defined as a simple constraint logic language, to simplify the formulation of abstract semantics for the language, and to indicate that semantics and algorithms may be generalized so as to apply to a broad class of CLP-languages [10].

$Pred$ is a collection of predicate symbols $p, q, \ldots$ of arity ≥ 0. $Atom$ contains atoms of the form $p(T_1 \ldots T_n)$, where $p \in Pred$ and $T_i \in Term$ is a term built in the usual way from function symbols $f, g, \ldots$ of arity ≥ 0, constants $a, b, \ldots$, and variables $X, Y, \ldots \in Var$. The function $vars : Term \rightarrow \mathcal{P}\ Var$ returns the set of variables occuring in a term. Con contains all equations of the form $A = A'$, where $A, A' \in Atom$, and the trivial constraint $true$. Con is closed under conjunction.

A constraint $\pi \in Con$ is satisfiable if it has a solution. A solution θ to π is a ground substitution (see [16]) such that the constraint $\theta\pi$ obtained from applying θ to π is true. Constraint $\theta(A = A')$ is true if θA and $\theta A'$ are identical. Constraint $\theta(\pi \wedge \pi')$ is true if $\theta\pi$ and $\theta\pi'$ are both true.

A program $P \in Progr$ is a collection of clauses. A clause $C \in Clause$ has the form $H :- \pi, B$, where $H \in Atom$, $\pi \in Con$, and $B \in Atom^*$. (The demand that constraints precede atoms in a clause body is made merely to simplify the presentation).

In the sequel we will refer to the standard top-down and bottom-up semantics of a program. We indicate how these notions can be defined:

The *standard top-down semantics* can be defined in terms of query rewriting. Query $\langle A :: G, \pi\rangle$ is rewritten, using renamed clause $H :- \pi', B$, into query $\langle B :: G, (\pi \wedge (A = H) \wedge \pi')\rangle$, if constraint $\pi \wedge (A = H) \wedge \pi'$ is satisfiable. The standard top-down semantics maps the initial query $\langle A, \pi\rangle$ to the set of all pairs $\langle A, \pi_{ans}\rangle$ such that query $\langle nil, \pi_{ans}\rangle$ can be obtained by query rewriting from the initial query.

The *standard bottom-up semantics* of a program is the set of all pairs $\langle A, \pi_{ans}\rangle$ associated by the standard top-down semantics to some initial query. The set will be referred to as the *success set* associated with the program.

Note that the notions of top-down and bottom-up semantics are unrelated to the "direction" of the fixpoint computation, which is bottom-up in all algorithms discussed in the paper.

The algorithms and abstract semantics defined in subsequent sections are generic in the sense that they accept a collection of externally given abstract domains and operators. As an example type of analysis we define a simple *groundness analysis*. A term T is said to be ground if $(vars\ T) = \emptyset$. Constraint π grounds term T if for all solutions θ to π we have that the term $\theta\pi$ is ground. For instance, constraint $p(X, Y) = p(a, Z)$ grounds X but neither Y nor Z. The groundness analysis is defined in terms of two abstract domains (both complete lattices) and four abstract operators (all monotonic).

The first abstract domain is Pat which consists of a number of so-called patterns for each predicate $p \in Pred$. Each argument to a predicate is described by one of the modes G (for ground) and A (for any).

Definition 3.1 For a given predicate $p \in Pred$ with arity k, the set Pat_p is given by

$$Pat_p = \{p(m_1 \ldots m_k) \mid m_i \in \{\mathsf{G}, \mathsf{A}\}\}$$

The set Pat of all patterns is given by $Pat = (\cup_{p \in Pred}\ Pat_p) \cup \{\perp, \top\}$.
The ordering relation $\sqsubseteq$ on Pat is defined by

$$p(m_1 \ldots m_k) \sqsubseteq p'(m'_1 \ldots m'_k) \ \Leftrightarrow\ \big(p = p' \text{ and } (\forall i : m_i = \mathsf{G} \text{ or } m'_i = \mathsf{A})\big)$$

□

The second abstract domain is $Acon$. An abstract constraint $\Pi \in Acon$ is a finite set of variables.

Definition 3.2 The set $Acon$ of abstract constraint is given by $Acon = \{\Pi \in \mathcal{P}\,Var \mid \Pi \text{ is finite}\} \cup \{\bot\}$ and is ordered by the superset relation. □

Note that $Acon$ has top element $\emptyset$.

The four abstract operators are defined as follows.

Definition 3.3 The four abstract operators have functionalities

$$
\begin{array}{lll}
add & : & Con \to Acon \to Acon \\
meet & : & Acon \to Acon \to Acon \\
project & : & Atom \to Acon \to Pat \\
generate & : & Atom \to Pat \to Acon
\end{array}
$$

and are defined by the equations:

$$
\begin{array}{lll}
add\ \pi\ W & = & W \cup \{V \mid \pi \text{ grounds } V\} \\
meet\ W\ W' & = & W \cup W' \\
project\ p(T_1 \ldots T_k)\ W & = & p(m_1 \ldots m_k) \text{ where } m_i = \begin{cases} \mathsf{G} & \text{if } (vars\ T_i) \subseteq W \\ \mathsf{A} & \text{otherwise} \end{cases} \\
generate\ p(T_1 \ldots T_k)\ p(m_1 \ldots m_k) & = & W_1 \cup \ldots \cup W_k \text{ where } W_i = \begin{cases} vars\ T_i & \text{if } m_i = \mathsf{G} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

with $meet\ \bot\ W = meet\ W\ \bot = add\ \pi\ \bot = generate\ p(T_1 \ldots T_n)\ \bot = \bot$
and $project\ p(T_1 \ldots T_n)\ \bot = \bot$ □

Example 4.2 below indicates how the operators can be combined in the analysis of a clause. A constant $\tau \in Acon$ is used to describe the constraint $true$. τ is given by $\tau = generate\ q\ q = \emptyset$, with q being any zero-ary predicate.

A function $pred$ return predicate symbols of atoms, patterns, and clauses:

$$
\begin{array}{llll}
(pred\ A) = p & \text{if} & A = p(T_1 \ldots T_k) & (A \in Atoms) \\
(pred\ \Delta) = p & \text{if} & \Delta = p(m_1 \ldots m_k) & (\Delta \in Pat) \\
(pred\ C) = p & \text{if} & C = (\,p(T_1 \ldots T_k) :- \pi, B\,) & (C \in Clause)
\end{array}
$$

In the complexity analysis the following notation is used:
$Atoms_C$: the number of atoms in clause C, couting the head as well as the body atoms.
$DomSize_C$: the size of Pat_p where $p = (pred\ C)$ is the predicate defined by clause C.
$DomHeight_C$: the height of the largest set $Pat_p \cup \{\bot\}$ for any predicate p called in the body of C.

4 Static dependency and bottom-up analysis

The bottom-up algorithm defined and analyzed in this section computes the abstract bottom-up semantics $\mathbf{P}_{\mathrm{bu}}\,[\![P]\!]$ of a given program P. The semantics $\mathbf{P}_{\mathrm{bu}}$ is defined formally, and the algorithm is proved correct with respect to the semantics. The simple nature of the bottom-up semantics allows for a chaotic iteration strategy based on a static dependency relation.

The abstract bottom-up semantics associates a single pattern with each predicate.

Definition 4.1 The semantic domain Sem_{bu} is defined by $Sem_{bu} = Pred \to Pat$. □

The intuitive idea is that the pattern associated with a predicate p is a *success* pattern. That is, the purpose of the abstract bottom-up semantics is to provide an approximation of the standard bottom-up semantics in the following sense: Let $S \in Sem_{bu}$ be the abstract bottom-up semantics of a given program, and suppose that $S\ p = p(m_1 \ldots m_k)$ for some predicate p. Then for any pair $\langle p(T_1 \ldots T_k), \pi) \rangle$ belonging to the success set of the program it must hold that constraint π grounds the i'th argument T_i whenever $m_i = \mathsf{G}$. A formal definition and proof of this property of $\mathbf{P}_{\mathrm{bu}}$ is outside the scope of the paper.

Definition 4.2 The abstract bottom-up semantics $\mathbf{P}_{\text{bu}}$ is defined as follows:
The semantic functions have functionalities:

$$\begin{array}{lcl} \mathbf{P}_{\text{bu}} & : & Progr \rightarrow Sem_{bu} \\ \mathbf{C}_{\text{bu}} & : & Clause \rightarrow Sem_{bu} \rightarrow Sem_{bu} \\ \mathbf{B}_{\text{bu}} & : & Atom^{*} \rightarrow Sem_{bu} \rightarrow Acon \rightarrow Acon \\ \mathbf{A}_{\text{bu}} & : & Atom \rightarrow Sem_{bu} \rightarrow Acon \rightarrow Acon \end{array}$$

and are defined by:

$$\begin{array}{lcl} \mathbf{P}_{\text{bu}}\,[\![P]\!] & = & lfp\,(\bigsqcup_{C \in P}\,(\mathbf{C}_{\text{bu}}\,[\![C]\!])) \\ \mathbf{C}_{\text{bu}}\,[\![H :- \pi, B]\!]\,S\,p & = & \text{if } (pred\ H) \not= p \text{ then } \bot \text{ else} \\ & & \quad \text{let } \Pi = \mathbf{B}_{\text{bu}}\,[\![B]\!]\,S\,(add\ \pi\ \tau) \text{ in } project\ H\ \Pi \\ \mathbf{B}_{\text{bu}}\,[\![nil]\!]\,S\,\Pi & = & \Pi \\ \mathbf{B}_{\text{bu}}\,[\![A :: B]\!]\,S\,\Pi & = & \text{let } \Pi' = \mathbf{A}_{\text{bu}}\,[\![A]\!]\,S\,\Pi \text{ in } \mathbf{B}_{\text{bu}}\,[\![B]\!]\,S\,\Pi' \\ \mathbf{A}_{\text{bu}}\,[\![A]\!]\,S\,\Pi & = & \text{let } \Pi' = generate\ A\,(S\,(pred\ A)) \text{ in } meet\ \Pi\,\Pi' \end{array}$$

□

$\mathbf{P}_{\text{bu}}$ is well-defined, due to completeness of Sem_{bu} and monotonicity of $\bigsqcup_{C \in P}$ ($\mathbf{C}_{\text{bu}}\,[\![C]\!]$) (cf. theorem 2.1).

Example 4.1 Consider the following simple Prolog program which will be referred to as P_{ex}.

(C_1) `p(a,Y):- true.`
(C_2) `p(X,Y):- true, p(Y,X).`

The abstract bottom-up semantics of the program is given by

$$\begin{array}{lcll} \mathbf{P}_{\text{bu}}\,[\![P_{ex}]\!]\,p & = & p(\mathsf{A}, \mathsf{A}) & \\ \mathbf{P}_{\text{bu}}\,[\![P_{ex}]\!]\,q & = & \bot & \text{if } q \not= p \end{array}$$

□

Stated in terms of the standard bottom-up semantics, the abstract semantics of program P_{ex} expresses that only queries to predicate p may possibly succeed, and that none of the arguments to p has been established to be always ground.

Example 4.2 Consider the evaluation of the expression $\mathbf{C}_{\text{bu}}\,[\![C_2]\!]\,S\,p$, where S satisfies $S\ p = p(\mathsf{G}, \mathsf{A})$. From the definition of clause C_2 (in example 4.1) it is intuitively clear that with the given S, the clause grounds the second argument to p. That is, one would expect the value of the expression to be $p(\mathsf{A}, \mathsf{G})$. To evaluate the expression we note that

$$\begin{array}{rcll} add\,true\,\tau & = & \emptyset & \text{(i)} \\ generate\ p(Y,X)\,p(\mathsf{G},\mathsf{A}) & = & \{Y\} & \text{(ii)} \\ meet\ \emptyset\,\{Y\} & = & \{Y\} & \text{(iii)} \\ project\ p(X,Y)\,\{Y\} & = & p(\mathsf{A},\mathsf{G}) & \text{(iv)} \end{array}$$

and obtain:

$$\begin{array}{lll} \mathbf{C}_{\text{bu}}\,[\![p(X,Y) :- true, p(Y,X)]\!]\,S\,p & & \\ \quad = \text{let } \Pi = \mathbf{B}_{\text{bu}}\,[\![p(Y,X)]\!]\,S\,(add\,true\ \tau) \text{ in} & & \text{(by definition of } \mathbf{C}_{\text{bu}}) \\ \qquad project\ \Pi\ p(X,Y) & & \\ \quad = \text{let } \Pi = \mathbf{A}_{\text{bu}}\,[\![p(Y,X)]\!]\,S\,\emptyset \text{ in } project\ p(X,Y)\,\Pi & & \text{(by (i) and definition of } \mathbf{B}_{\text{bu}}) \\ \quad = \text{let } \Pi' = generate\ p(Y,X)\,p(\mathsf{G},\mathsf{A}) \text{ in} & & \text{(by definition of } \mathbf{A}_{\text{bu}}) \\ \qquad \text{let } \Pi = meet\ \emptyset\,\Pi' \text{ in } project\ p(X,Y)\,\Pi & & \\ \quad = \text{let } \Pi = meet\ \emptyset\,\{Y\} \text{ in } project\ p(X,Y)\,\Pi & & \text{(by (ii))} \\ \quad = project\ p(X,Y)\,\{Y\} & & \text{(by (iii))} \\ \quad = p(\mathsf{A},\mathsf{G}) & & \text{(by (iv))} \end{array}$$

□

Now we turn to the computation of $\mathbf{P}_{\mathrm{bu}}\,[\![P]\!]$ for a given input program P. A brute-force approach is suggested by proposition 2.2. One may simply compute the ascending Kleene sequence for $\bigsqcup_{C \in P}\,(\mathbf{C}_{\mathrm{bu}}\,[\![C]\!])$ given by

$$\begin{array}{lcl} S^0 & = & \lambda p.\bot \\ S^n & = & \bigsqcup_{C \in P}\,(\mathbf{C}_{\mathrm{bu}}\,[\![C]\!])\,S^{n-1} \end{array}$$

where $S^n \in Sem_{bu}$. At the point where $S^{n+1} = S^n$ we have $S^n = lfp\,(\bigsqcup_{C \in P}\,(\mathbf{C}_{\mathrm{bu}}\,[\![C]\!]))$.

The notion of chaotic iteration was introduced by Cousot and Cousot [7] [8]. Chaotic iteration may be applied in the present context to yield the following improvement over the brute-force method: Immediately upon evaluation of $\mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\,S\,p$, the result is "added" to the iterand S. By contrast, in the brute-force method, all clauses in the program are scanned once using a fixed iterand before the iterand is modified. Chaotic iteration yields a sequence

$$\begin{array}{lcl} S^0 & = & \lambda p.\bot \\ S^n & = & S^{n-1} \sqcup (\mathbf{C}_{\mathrm{bu}}\,[\![C_n]\!]\,S^{n-1}) \end{array}$$

for some sequence of clauses $C_1, \ldots, C_n, \ldots$. Several methods exist to secure that a chaotic iteration sequence eventually stabilizes at $\bigsqcup_{C \in P}\,(\mathbf{C}_{\mathrm{bu}}\,[\![C]\!])$. Perhaps the most simple is to select clauses according to their order in the program, and stop when all clauses in P has been tried and seen not to change the iterand.

The bottom-up algorithm can be seen as choosing a chaotic iteration strategy based on a static dependency relation. The strategy prescribes that if the n'th iteration leads to an *update* of the pattern for predicate p, i.e., if $(S^{n-1}p) \sqsubset (S^n\,p)$, then the clauses that *depend* on p should be selected for evaluation. Clause C depends on predicate p if for some atom B_i in the body of C we have $(pred\ \ B_i) = p$.

The observation underlying this approach is that if clause C does not depend on the updated predicate p, then there is no point in (re)evaluating C. This is expressed in the following proposition, which is easily proved by induction over the number of atoms in the body of a given clause.

Proposition 4.1 *Let C be the clause $(H :\!-\ \pi, B_1 \ldots B_m)$, with $(pred\ B_i) = p_i$. Let $S, S' \in Sem_{bu}$ satisfy $S' = \mathbf{C}_{\mathrm{bu}}\,[\![C']\!]\,S$ for some clause C'. Then $(\forall i \in [1, m] : S'\,p_i = S\,p_i) \Rightarrow \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\,S' = \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\,S$.*

The relation between predicates and the clauses they depend on is static in the sense that the relation can be computed once and for all prior to the fixpoint iteration. The dependency information can be obtained on a syntactic basis in the sense that only the predicate symbols occuring in a clause body need be examined, while it is unnecessary to consider for instance the definition of the abstract operators.

The bottom-up algorithm is defined as follows in pseudo-code.

Input: Program P.
Output: The value of S upon termination of the algorithm is $\mathbf{P}_{\mathrm{bu}}\,[\![P]\!]$.

```
%%  initialization
workset := all clauses in P
for all predicates p defined in program P do
  S(p) := ⊥
  depend(p) = {C ∈ P | p is called in body of C} od
%%  iteration
while workset ≠ ∅ do
  select and delete clause C from workset
  let (H :- π, B) = C  and  p = (pred H)
  Δ := C_bu [[C]] S p
  if Δ ⋢ S(p) then
    S(p) := S(p) ⊔ Δ
    workset := workset ∪ depend(p) fi
od
```

The definition is in terms of the following variables ranging over composite domains:

The variable $S \in Sem_{bu}$ represents the iterands. Initially, S has the bottom value $\lambda p.\bot$. Upon termination, S holds the algorithm output $\mathbf{P}_{\mathrm{bu}}\,[\![P]\!]$.

The variable $workset \in (\mathcal{P}\ Clause)$ holds the set of clauses that must be scanned in the following iterations. Initially, $workset$ contains all clauses in the given program. The algorithm terminates when $workset$ is empty.

The variable $depend \in (Pred \to \mathcal{P}\ Clause)$ represents the dependency information. For a given predicate q defined in P, clauses C belongs to $depend(q)$ if the clause contains a call to q in the body.

The variables ranging over simple domains are $P \in Progr$, $C \in Clause$, $p \in Pred$, and $\Delta \in Pat$.

The questions of what data structures should be used to represent the variables, and how the various operations on them should be implemented, are not considered. Of course, the values of $S\ q$ and $depend(q)$ are irrelevant if q is not defined by a clause in the input program.

In the following analysis of the algorithm, an iteration refers to an execution of the while-loop. S^n and $workset^n$ denote the values of S and $workset$ upon the n'th iteration (say, at the point of the test $workset \neq \emptyset$). Note that $S^{n-1} \sqsubseteq S^n$ always holds, since S is only updated "upwards". We refer to the evaluation of an expression $\mathbf{C}_{\mathrm{td}}\, [\![C]\!]\, S\, p$ (for given C, S, and $p = (pred\ C)$) as a *clause scan*.

Example 4.3 Consider program P_{ex} from example 4.1 as input to the bottom-up algorithm. It is assumed that clauses are added to the $workset$ according to their order in the program, and deleted from the $workset$ on a first-in first-out basis.

At initialization, the variable $depend$ is assigned the function

$$\begin{array}{lcll} depend(p) & = & \{C_2\} & \\ depend(q) & = & \emptyset & \text{if } q \neq p \end{array}$$

while S and $workset$ are initialized as shown in the top row of the table. The bottom-up algorithm performs three iterations:

n	Clause scan	S^n	$workset^n$
0		$p \to \bot$	$\{C_1, C_2\}$
1	$\mathbf{C}_{\mathrm{bu}}\, [\![C_1]\!]\, S^0\ p = p(\mathsf{G}, \mathsf{A})$	$p \to p(\mathsf{G}, \mathsf{A})$	$\{C_2\}$
2	$\mathbf{C}_{\mathrm{bu}}\, [\![C_2]\!]\, S^1\ p = p(\mathsf{A}, \mathsf{G})$	$p \to p(\mathsf{A}, \mathsf{A})$	$\vdots$
3	$\mathbf{C}_{\mathrm{bu}}\, [\![C_2]\!]\, S^2\ p = p(\mathsf{A}, \mathsf{A})$	$\vdots$	$\emptyset$

A field with ($\vdots$) has the same value as the field above it. The value $p \to p(\mathsf{G}, \mathsf{A})$ of the field for S^1 stands for the function that maps p to $p(\mathsf{G}, \mathsf{A})$ and other predicates to $\bot$.

In the first iteration, clause C_1 is selected. Evaluation of $\mathbf{C}\, [\![C_1]\!]\, S^0\ p$ leads to an update of p, and so $\{C_2\}$ is added to $workset$, which however already contains the clause. The second iteration leads to another update of p; this iteration corresponds to the computation detailed in example 4.2. The success pattern $p(\mathsf{A}, \mathsf{A}) = (S^2\ p)$ is obtained as the least upper bound of the set $\{p(\mathsf{G}, \mathsf{A}), p(\mathsf{A}, \mathsf{G})\}$; the analysis misses out the information that either the first or the second argument is ground, because each predicate is described by only a single success pattern. The third iteration is the last one since it leaves $workset$ empty. $\square$

Correctness of the bottom-up algorithm is defined with respect to the abstract bottom-up semantics: For an arbitrary program P, the output of the algorithm as executed with input P must be equal to $\mathbf{P}_{\mathrm{bu}}\, [\![P]\!]$.

Proposition 4.2 *The bottom-up algorithm is correct.*

Proof Suppose the algorithm is executed with program P as input, terminating after N iterations. We must show that $S^N = \mathbf{P}_{\mathrm{bu}}\, [\![P]\!]$.

$S^N \sqsubseteq \mathbf{P}_{\mathrm{bu}}\, [\![P]\!]$: Since for all clauses C we have $\mathbf{C}_{\mathrm{bu}}\, [\![C]\!] \preceq (\sqcup_{C \in P}\, \mathbf{C}_{\mathrm{bu}}\, [\![C]\!])$, the inclusion follows from proposition 2.3, point (i).

$\mathbf{P}_{\mathrm{bu}}\,[\![P]\!] \sqsubseteq S^N$: By proposition 2.3 (ii) it suffices to show

$$\bigsqcup_{C \in P} (\mathbf{C}_{\mathrm{bu}}\,[\![C]\!])\, S^N \quad \sqsubseteq \quad S^N \tag{1}$$

To show (1), it suffices to establish the invariant

$$\forall n \in [0, N]\, \forall C:\; C \notin workset^n \Rightarrow \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^n \sqsubseteq S^n \tag{2}$$

since the workset is empty at termination. To establish (2), we proceed by induction over n.
Base case: For $n = 0$ the invariant holds because initially $workset$ contains all clauses of P.
Inductive case: Let C be an arbitrary clause. By induction hypothesis we have $C \notin workset^{n-1} \Rightarrow \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^{n-1} \sqsubseteq S^{n-1}$, and we must show $C \notin workset^n \Rightarrow \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^n \sqsubseteq S^n$. We use that if clause C is not added to *workset* upon the n'th iteration, the equalities $(S^n\; p_i) = (S^{n-1}\; p_i)$ hold for all predicates p_i called in the body of C. Thus by proposition 4.1 we have

$$C \notin workset^n \quad \Rightarrow \quad \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^n = \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^{n-1} \tag{3}$$

We distinguish between two cases:
(a) $C \in workset^{n-1}$ and $C \notin workset^n$. In this case C must be the clause which is scanned in the n'th iteration. Thus we have $S^n = S^{n-1} \sqcup (\mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^{n-1})$. By substituting into this equation the equation implied in (3) we obtain $S^n = S^{n-1} \sqcup (\mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^n)$, which establishes $\mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^n \sqsubseteq S^n$.
(b) $C \notin workset^{n-1}$ and $C \notin workset^n$: Using equation (3) and the induction hypothesis we have $\mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^n = \mathbf{C}_{\mathrm{bu}}\,[\![C]\!]\, S^{n-1} \sqsubseteq S^{n-1} \sqsubseteq S^n$. $\Box$

The complexity of the bottom-up algorithm is expressed in terms of the number of scans of each clause in a given program, i.e., the number of passes of dataflow over each clause.

Proposition 4.3 *The bottom-up algorithm performs at most $Atoms_C * DomHeight_C$ scans of any clause C in any program on which the algorithm is run.*

Proof Let a program P be given, and let C be a clause in it. The clause is scanned once for every time it is added to *workset*. The clause is added to *workset* initially, plus every time there is an update for some predicate called in the body of the clause. There are $Atoms_C - 1$ atoms in the body of C, and each of the corresponding predicates can be updated at most $DomHeight_C$ times. Thus, the number of scans of clause C is bounded by $1 + DomHeight_C * (Atoms_C - 1) \leq DomHeight_C * Atoms_C$ (since $DomHeight_C \geq 1$). $\Box$

The bottom-up algorithm's iteration strategy, being based on a static dependency relation, is similar to the strategies underlying the algorithms proposed by for instance Kildall [17], O'Keefe [15], and Debray [9]. Significantly, by the above complexity result, the number of passes of dataflow analysis over each clause in a program is independent of the remainder of the program, i.e., of any measure on the size of the rest of the program. This is the property that we would like to carry over to the more difficult problem which arises when a distinction is made between different calls to the same predicate.

5 Dynamic dependency and top-down analysis

The top-down algorithm defined and analyzed in this section computes the abstract top-down semantics $\mathbf{P}_{\mathrm{td}}\,[\![P]\!]$ of a given program P. The abstract top-down semantics of a program may express more information about the program than the abstract bottom-up semantics. On the other hand, it is more difficult to compute. The top-down algorithm copes with the more difficult problem by employing a chaotic iteration strategy which can be said to be based on a dynamic dependency relation.

Definition 5.1 The semantic domain Sem_{td} is defined by $Sem_{td} = Pat \to Pat$. $\Box$

The abstract top-down semantics is intended to approximate the standard top-down semantics in the following sense: Let $F \in Sem_{td}$ be the abstract top-down semantics of a program, let $p(m_1 \dots m_k)$ be an arbitrary pattern, and let $\langle p(T_1 \dots T_k), \pi\rangle$ be an arbitrary initial query to the program. Suppose that constraint π grounds T_i whenever m_i = G. Then for all pairs $\langle p(T'_1 \dots T'_k), \pi_{ans}\rangle$ associated by the standard top-down semantics to the initial query it must hold that π_{ans} grounds T'_i whenever m'_i = G, with $p(m'_1 \dots m'_k) = F\ p(m_1 \dots m_k)$.

Definition 5.2 The top-down semantics $\mathbf{P}_{\mathrm{td}}$ is defined as follows:
The semantic functions have functionalities:

$$\begin{array}{lcl}
\mathbf{P}_{\mathrm{td}} & : & Progr \to Sem_{td} \\
\mathbf{C}_{\mathrm{td}} & : & Clause \to Sem_{td} \to Sem_{td} \\
\mathbf{B}_{\mathrm{td}} & : & Atom^* \to Sem_{td} \to Acon \to Acon \\
\mathbf{A}_{\mathrm{td}} & : & Atom \to Sem_{td} \to Acon \to Acon
\end{array}$$

and are defined by:

$$\begin{array}{lcl}
\mathbf{P}_{\mathrm{td}}\ [\![P]\!] & = & lfp\ (\bigsqcup_{C \in P}\ (\mathbf{C}_{\mathrm{td}}[\![C]\!])) \\
\mathbf{C}_{\mathrm{td}}\ [\![H :- \pi, B]\!]\ F\ \Delta & = & \text{let } \Pi = generate\ H\ \Delta \text{ in} \\
 & & \quad \text{let } \Pi' = add\ \pi\ \Pi \text{ in} \\
 & & \quad\quad \text{let } \Pi'' = \mathbf{B}_{\mathrm{td}}\ [\![B]\!]\ F\ \Pi' \text{ in } \ project\ H\ \Pi'' \\
\mathbf{B}_{\mathrm{td}}\ [\![nil]\!]\ F\ \Pi & = & \Pi \\
\mathbf{B}_{\mathrm{td}}\ [\![A :: B]\!]\ F\ \Pi & = & \mathbf{B}_{\mathrm{td}}\ [\![B]\!]\ F\ (\mathbf{A}_{\mathrm{td}}\ [\![A]\!]\ F\ \Pi) \\
\mathbf{A}_{\mathrm{td}}\ [\![A]\!]\ F\ \Pi & = & \text{let } \Delta = project\ A\ \Pi \text{ in} \\
 & & \quad \text{let } \Pi' = generate\ A\ (F\ \Delta) \text{ in } \ meet\ \Pi\ \Pi'
\end{array}$$

□

$\mathbf{P}_{\mathrm{td}}$ is well-defined, due to completeness of Sem_{td} and monotonicity of $\bigsqcup_{C \in P}$ ($\mathbf{C}_{\mathrm{td}}$ $[\![C]\!]$) (cf. theorem 2.1).

Example 5.1 The abstract top-down semantics of program P_{ex} (see example 4.1) is given by

$$\begin{array}{lcll}
\mathbf{P}_{\mathrm{td}}\ [\![P_{ex}]\!]\ p(\mathsf{G},\mathsf{G}) & = & p(\mathsf{G},\mathsf{G}) & \\
\mathbf{P}_{\mathrm{td}}\ [\![P_{ex}]\!]\ p(\mathsf{G},\mathsf{A}) & = & p(\mathsf{G},\mathsf{A}) & \\
\mathbf{P}_{\mathrm{td}}\ [\![P_{ex}]\!]\ p(\mathsf{A},\mathsf{G}) & = & p(\mathsf{A},\mathsf{G}) & \\
\mathbf{P}_{\mathrm{td}}\ [\![P_{ex}]\!]\ p(\mathsf{A},\mathsf{A}) & = & p(\mathsf{A},\mathsf{A}) & \\
\mathbf{P}_{\mathrm{td}}\ [\![P_{ex}]\!]\ q(m_1 \dots m_k) & = & \bot & \text{if } q \neq p
\end{array}$$

□

The difference between the abstract top-down and bottom-up semantics is that the former takes call patterns into account: First, when a clause is entered, an abstract constraint is "generated" which may possibly ground certain variables occuring in the clause head, due to the calling pattern for the clause. Second, during scanning of a clause, when a body atom is to be evaluated, a call pattern for the atom is computed by "projecting" the information contained in the abstract constraint onto the atom. The two semantics are otherwise closely related. In fact, if the top-down version had operated on a domain which allowed only for a single call pattern per predicate, the two semantics would have been equivalent.

To compute $\mathbf{P}_{\mathrm{td}}$ $[\![P]\!]$ for a given program P, one may proceed in a brute-force manner, constructing the ascending Kleene sequence for the function $\bigsqcup_{C \in P}$ ($\mathbf{C}_{\mathrm{td}}[\![C]\!]$).

To improve efficiency, one may use some form of chaotic iteration. We first define the content of an iteration, and then consider the over-all iteration strategy.

An iteration computes iterand $F^n \in Sem_{td}$ from the previous one in three steps:

Step one: A clause scan. That is, the evaluation of the expression $\mathbf{C}_{\mathrm{td}}$ $[\![C_n]\!]$ F^{n-1} Δ_n, with $\langle C_n, \Delta_n\rangle$ being the pair chosen for the n'th iteration.
Step two: The result of the clause scan is added to the iterand:

$$F = F^{n-1} \sqcup \mathbf{S}(C_n, F^{n-1}, \Delta_n) \quad \text{with } \mathbf{S}(C, F, \Delta) = \lambda\Delta'.(\text{if } \Delta' \neq \Delta \text{ then } \bot \text{ else } \mathbf{C}_{\mathrm{td}}\ [\![C]\!]\ F\ \Delta)$$

Step three: Monotonicity of the iterand is preserved. By steps one and two, at most a single pattern Δ_{upd} will have been updated, i.e., $(F^{n-1}\Delta_{upd}) \sqsubset (F\,\Delta_{upd})$; and there may exist patterns Δ where $\Delta_{upd} \sqsubset \Delta$ and $(F\Delta_{upd}) \not\sqsubseteq (F\Delta)$. Working with monotonic iterands is intended to speed up convergence by assigning $(F\ \Delta)$ the larger value $(F\ \Delta) \sqcup (F\ \Delta_{upd})$. Step three may be expressed in terms of the function **M** defined by $\mathbf{M}(F) = \lambda\Delta.(\sqcup\,\{F\ \Delta' \mid \Delta' \sqsubseteq \Delta\})$.

Chaotic iteration based on steps one through three yields a sequence

$$\begin{array}{lcl} F^0 & = & \lambda\Delta.\bot \\ F^n & = & \mathbf{M}(F^{n-1} \sqcup\ \mathbf{S}(C_n, F^{n-1}, \Delta_n)) \end{array}$$

for a sequence of pairs $\langle C_n, \Delta_n\rangle$. The problem then is to find a good strategy for selecting the sequence of pairs. That is, a strategy which ensures quick convergence to the least fixpoint, and in which the determination of the sequence of pairs is itself not too expensive. Of course, we should only select pairs $\langle C, \Delta\rangle$ where $(pred\ C) = (pred\ \Delta)$, since otherwise we trivially have that $\mathbf{C}_{\mathrm{td}}\ [\![C]\!]\ F\ \Delta = \bot$ (for all F).

A straightforward generalization of the method underlying the bottom-up algorithm of section 4 is to define a dependency relation as follows:

The pair $\langle C, \Delta\rangle$ depends on Δ_{upd} if there is an atom B_i in the body of C such that $(pred\ B_i) = (pred\ \Delta_{upd})$, with Δ being any pattern satisfying $(pred\,\Delta) = (pred\,C)$. As an example, consider clause C_2 in program P_{ex} from example 4.1:

```
(C2)    p(X,Y):- true, p(Y,X).
```

This dependency relation would lead to the following: If any pattern for predicate p is updated, then all four tuples $\langle C_2, p(\mathsf{G},\mathsf{G})\rangle$, $\langle C_2, p(\mathsf{G},\mathsf{A})\rangle$, $\langle C_2, p(\mathsf{A},\mathsf{G})\rangle$, and $\langle C_2, p(\mathsf{A},\mathsf{A})\rangle$ should be reprocessed.

However, this approach seems to be too coarse. It appears to be possible to achieve a more efficient analysis of program P_{ex} by relying on a more fine-grained dependency relation where

$$\begin{array}{lcl} \langle C_2, p(\mathsf{G},\mathsf{A})\rangle & \text{depends on} & \langle p(\mathsf{A},\mathsf{G})\rangle \\ \langle C_2, p(\mathsf{A},\mathsf{G})\rangle & \text{depends on} & \langle p(\mathsf{G},\mathsf{A})\rangle \\ & etc. & \end{array}$$

Defining a more fine-grained dependency relation leads to the consideration of a dynamic relation, that is, one which is modified during the fixpoint iteration. As an example, consider the following program:

```
(C1)    p(a,Y):- true.
(C2)    p(X,Y):- true, p(Y,X), q(Y,X).
(C3)    q(X,Y):-    ...
```

Suppose an iterand F^n in a chaotic sequence for the program satisfies $F^n\,p(\mathsf{A},\mathsf{A}) = p(\mathsf{G},\mathsf{A})$. If we in the n+1'st iteration evaluate $\mathbf{C}_{\mathrm{td}}\ [\![C_2]\!]\ F^n\ p(\mathsf{A},\mathsf{A})$, the call pattern $q(\mathsf{G},\mathsf{A})$ will be computed for $q(Y, X)$. Thus, we are led to consider $\langle C_2, p(\mathsf{A},\mathsf{A})\rangle$ as depending on $q(\mathsf{G},\mathsf{A})$. Now, suppose a subsequent iterand F^{n+d} satisfies $F^{n+d}\ p(\mathsf{A},\mathsf{A}) = p(\mathsf{A},\mathsf{A})$. That would lead us to consider $\langle C_2, p(\mathsf{A},\mathsf{A})\rangle$ as depending on $q(\mathsf{A},\mathsf{A})$. Thus given the above program we obtain:

$$\begin{array}{llll} \langle C_2, p(\mathsf{A},\mathsf{A})\rangle & \text{depends on} & \langle p(\mathsf{A},\mathsf{A})\rangle \text{ and } \langle q(\mathsf{G},\mathsf{A})\rangle & \text{upon iteration } (n+1) \\ \langle C_2, p(\mathsf{A},\mathsf{A})\rangle & \text{depends on} & \langle p(\mathsf{A},\mathsf{A})\rangle \text{ and } \langle q(\mathsf{A},\mathsf{A})\rangle & \text{upon iteration } (n+d+1) \\ & etc. & & \end{array}$$

These considerations motivate the definition of a dynamic dependency relation, where $\langle C, \Delta\rangle$ depends on Δ' if Δ' has been computed as a call pattern for an atom in the body of C, during the most recent evaluation of Δ against C.

The usefulness of a dynamic dependency relation can be traced to the definition of the semantic function $\mathbf{A}_{\mathrm{td}}$ (in definition 5.2 of the abstract top-down semantics). To evaluate $\mathbf{A}_{\mathrm{td}}\ [\![A]\!]\ F\ \Pi$, the function F must be applied to $\Delta = (project\ A\ \Pi)$, where the abstract constraint Π is the result of processing the part of the clause to the left of atom A, implying that Π depends on F. Thus, the value of $\Delta = (project\ A\ \Pi)$ cannot (at a reasonably low cost) be predicted prior to the fixpoint iteration. By contrast, definition 4.2 of the bottom-up semantics prescribes that the evaluation of

Input: Program P.
Output: The value of F upon termination of the algorithm is $\mathbf{P}_{\mathrm{td}}\,[\![P]\!]$.

```
%%  initialization
workset := {⟨C, Δ⟩ | C ∈ P and (pred C) = (pred Δ)}
for all relevant Δ  do F := ⊥                              % Δ  is relevant if (pred Δ)
for all relevant Δ  do depend(Δ) = ∅ od                    % is defined in P.
for all relevant ⟨C, Δ, i⟩ do last_call(C, Δ, i) = ⊥ od    % i  is relevant if 1 ≤ i < Atoms_C.
for all relevant Δ do fathers(Δ) = {Δ′ ∈ Pat_(pred Δ) | Δ ⊏ Δ′ and (¬∃Δ″ : Δ ⊏ Δ″ ⊏ Δ′)} od
%%  iteration
while workset ≠ ∅ do
  select and delete ⟨C, Δ_call⟩ from workset
  Δ_succ := scan(C, F, Δ_call)
  if Δ_succ ⋢ F(Δ_call) then update(Δ_call, Δ_succ) fi
od

procedure update(Δ_call, Δ_succ)
begin
  F(Δ_call) := F(Δ_call) ⊔ Δ_succ
  for all ⟨C, Δ, i⟩ ∈ depend(Δ_call) do workset := workset ∪ {⟨C, Δ⟩} od
  for all Δ ∈ fathers(Δ_call) do   if Δ_succ ⋢ F(Δ) then update(Δ, Δ_succ) fi od
end

function scan(C, F, Δ_call)
begin
  let (H :− π, B_1 ... B_m) = C
  Π := generate(H, Δ_call)
  Π := add(π, Π)
  for i := 1 ... m do
    Δ_new := project(B_i, Π)
    Π′ := generate(B_i, F(Δ_new))
    Π := meet(Π, Π′)
    Δ_old := last_call(C, Δ_call, i)
    last_call(C, Δ_call, i) := Δ_new
    depend(Δ_old) := depend(Δ_old) \ {⟨C, Δ_call, i⟩}       % modifying the dynamic
    depend(Δ_new) := depend(Δ_new) ∪ {⟨C, Δ_call, i⟩}       % dependency relation
  od
  Δ_succ := project(H, Π)
  return Δ_succ
end
```

Figure 1: The top-down algorithm based on dynamic dependency information. The variables ranging over simple domains are $P \in Progr$, $C \in Clause$, $H, B_i \in Atom$, $\pi \in Con$, $\Pi \in Acon$, $\Delta, \Delta_{call}, \ldots \in Pat$, and $i \in \mathcal{N}$.

$\mathbf{A}_{\text{bu}}$ $[\![A]\!]$ S Π requires the application of S to the syntactically given entity ($pred\ A$).

The top-down algorithm is specified in figure 1 in pseudo-code. The main program is divided into an initialization and an iteration part. In the latter, procedures $update$ and $scan$ are called. Procedure $scan$ implements step one. It is a function which returns the value of an expression $\mathbf{C}_{\text{td}}$ $[\![C]\!]$ F Δ and as a side-effect modifies the dynamic dependency information. Procedure $update$ implements steps two and three. Monotonicity of the iterand is preserved by recursively working upwards in the lattice of patterns for the predicate in question. The procedure also takes care of adding pairs to *workset*.

The variables ranging over composite domains are:

The variable $F \in Sem_{td}$ represents the iterand. Initially, F is assigned the bottom value $\lambda\Delta.\bot$. Upon termination F is the least fixpoint of $\bigsqcup_{C \in P}$ ($\mathbf{C}_{\text{td}}$ $[\![C]\!]$).

The variable $workset \in \mathcal{P}\ (Clause \times Pat)$ holds the pairs $\langle C, \Delta \rangle$ that are waiting to be processed.

The variable $depend \in (Pat \rightarrow \mathcal{P}\ (Clause \times Pat \times \mathcal{N}))$ maps each pattern Δ to the set of tuples $\langle C, \Delta', i \rangle$ that are currently depending on Δ. The index i refers to an atom in the body of C, namely the atom for which call pattern Δ was computed during a previous scan of C with input pattern Δ'.

The variable $last\_call \in (Clause \times Pat \times \mathcal{N}) \rightarrow Pat$ is continuously modified so that the value of $last\_call(C, \Delta, i)$ is always the call pattern which has been computed for the i'th body atom of C, during the most recent evaluation of Δ against C. Initially, $last\_call$ maps all triples to $\bot$. Upon each iteration, $last\_call$ is the inverse function of $depend$ (see the invariant (4) below). The variable serves as a means of deciding in which points the dependency relation should be modified during a clause scan.

The variable $fathers \in (Pat \rightarrow \mathcal{P}\ Pat)$ is assigned a function which maps any pattern Δ to the set of patterns placed immediately above Δ, with respect to the ordering relation on Pat. This function is used by procedure $update$ in the preservation of monotonicity of the iterand.

In the analysis of the algorithm, an iteration refers to an execution of the while-loop controlled by the test $workset \neq \emptyset$. F^n, $depend^n(C, \Delta, i)$, etc., denote the values of the variables upon the n'th iteration. A running example is shown in Appendix A.

It can be shown that for all input programs P the following properties hold invariantly for all n:

$$\langle C, \Delta_{dep}, i \rangle \in depend^n(\Delta) \quad \Leftrightarrow \quad \Delta = last\_call^n(C, \Delta_{dep}, i) \tag{4}$$

$$last\_call^{n-1}(C, \Delta, i) \quad \sqsubseteq \quad last\_call^n(C, \Delta, i) \tag{5}$$

$$F^{n-1}(last\_call^{n-1}(C, \Delta, i)) \quad \sqsubseteq \quad F^n(last\_call^n(C, \Delta, i)) \tag{6}$$

Furthermore, we have that for $n \geq 1$

$$\begin{aligned} & \langle C, \Delta \rangle \text{ is added to } workset^n \\ \Leftrightarrow \quad & \exists i, \Delta_{upd} : (F^{n-1}\ \Delta_{upd}) \sqsubset (F^n\ \Delta_{upd}) \text{ and } \langle C, \Delta, i \rangle \in depends^n(\Delta_{upd}) \\ \Leftrightarrow \quad & \exists i : F^{n-1}(last\_call^n(C, \Delta, i)) \sqsubset F^n(last\_call^n(C, \Delta, i)) \qquad \text{(by invariant 4)} \end{aligned} \tag{7}$$

Since $F^{n-1} \preceq F^n$ we have that if $\langle C, \Delta \rangle \notin workset^n$ then the equation $F^{n-1}(last\_call^n(C, \Delta, i)) = F^n(last\_call^n(C, \Delta, i))$ holds for all $i < Atoms_C$. It follows that

$$\langle C, \Delta \rangle \notin workset^n \Rightarrow \mathbf{C}_{\text{td}}\ [\![C]\!]\ F^n\ \Delta = \mathbf{C}_{\text{td}}\ [\![C]\!]\ F^{n-1}\ \Delta \tag{8}$$

Correctness of the top-down algorithm is defined with respect to the abstract top-down semantics: For an arbitrary program P, the output of the algorithm as executed with input P must be equal to $\mathbf{P}_{\text{td}}$ $[\![P]\!]$. We assume that the algorithm implements iteration steps one through three as defined above.

Proposition 5.1 *The top-down algorithm is correct.*

Proof Suppose the algorithm is executed with an arbitrary program P as input, terminating after N iterations. We must show that $F^N = \mathbf{P}_{\text{td}}$ $[\![P]\!]$.

$F^N \sqsubseteq \mathbf{P}_{\text{td}}\ [\![P]\!]$: By proposition 2.3 (i) it suffices to show that $F^n \sqsubseteq (F^{n-1}) \sqcup (\bigsqcup_{C \in P} (\mathbf{C}_{\text{td}}\ [\![C]\!])\ F^{n-1})$ holds for all n. We use that F^n is monotonic for all n, and that $\mathbf{M}(F) = F$ holds if F is monotonic.

$$
\begin{array}{rcll}
F^n & = & \mathbf{M}((F^{n-1}) \sqcup (\mathbf{S}(C_n, F^{n-1}, \Delta_n))) & \text{(by definition of an iteration)} \\
& \sqsubseteq & \mathbf{M}((F^{n-1}) \sqcup (\bigsqcup_{C \in P} (\mathbf{C}_{\text{td}}\ [\![C]\!])\ F^{n-1})) & \text{(by definition of } \mathbf{S}\text{)} \\
& = & (F^{n-1}) \sqcup (\bigsqcup_{C \in P} (\mathbf{C}_{\text{td}}\ [\![C]\!])\ F^{n-1}) & \text{(by monot. of } F^{n-1} \text{ and } \bigsqcup_{C \in P} (\mathbf{C}_{\text{td}}\ [\![C]\!])\ F^{n-1}\text{)}
\end{array}
$$

$\mathbf{P}_{\text{td}}\ [\![P]\!] \sqsubseteq F^N$: By proposition 2.3 (ii) it suffices to show

$$
\bigsqcup_{C \in P} (\mathbf{C}_{\text{td}}\ [\![C]\!])\ F^N \sqsubseteq F^N \tag{9}
$$

To show (9) it suffices to establish the invariant

$$
\forall n \in [0, N]\, \forall C, \Delta : \langle C, \Delta \rangle \notin workset^n \Rightarrow \mathbf{C}_{\text{td}}\ [\![C]\!]\ F^n \Delta \sqsubseteq F^n\ \Delta
$$

since $workset$ is empty at termination. We proceed by induction over n:
Base case: For $n = 0$ the invariant holds because $workset$ initially contains all relevant pairs $\langle C, \Delta \rangle$.
Inductive case: We let C and Δ be an arbitrary clause and pattern. By induction hypothesis we have $\langle C, \Delta \rangle \notin workset^{n-1} \Rightarrow \mathbf{C}_{\text{td}}\ [\![C]\!]\ F^{n-1}\Delta \sqsubseteq (F^{n-1}\ \Delta)$, and we must show $\langle C, \Delta \rangle \notin workset^n \Rightarrow \mathbf{C}_{\text{td}}\ [\![C]\!]\ F^n \Delta \sqsubseteq (F^n\ \Delta)$. We distinguish between two cases:

(a) $\langle C, \Delta \rangle \in workset^{n-1}$ and $\langle C, \Delta \rangle \notin workset^n$.
In this case the pair $\langle C, \Delta \rangle$ must be the pair processed in the n'th iteration. Thus

$$
\begin{array}{rcll}
F^n & = & \mathbf{M}((F^{n-1}) \sqcup \mathbf{S}(C, F^{n-1}, \Delta)) & \text{(by definition of an iteration)} \\
& = & \mathbf{M}((F^{n-1}) \sqcup \mathbf{S}(C, F^n, \Delta)) & \text{(by (8))}
\end{array}
$$

which by definition of **S** implies $\mathbf{C}_{\text{td}}\ [\![C]\!]\ F^n\ \Delta \sqsubseteq F^n\ \Delta$.

(b) $\langle C, \Delta \rangle \notin workset^{n-1}$ and $\langle C, \Delta \rangle \notin workset^n$.
In this case we make use of the induction hypothesis:

$$
\begin{array}{rcll}
\mathbf{C}_{\text{td}}\ [\![C]\!]\ F^n\ \Delta & = & \mathbf{C}_{\text{td}}\ [\![C]\!]\ F^{n-1}\ \Delta & \text{(by (8))} \\
& \sqsubseteq & F^{n-1}\ \Delta & \text{(by induction hypothesis)} \\
& \sqsubseteq & F^n\ \Delta & \text{(since } F \text{ is only updated ``upwards'')}
\end{array}
$$

□

The complexity of the top-down algorithm is expressed in terms of the number of clause scans, i.e., evaluations of $\mathbf{C}_{\text{td}}\ [\![C]\!]\ F\ \Delta$.

Proposition 5.2 *The top-down algorithm performs at most $Atoms_C * DomHeight_C * DomSize_C$ scans of any clause C in any program on which the algorithm is run.*

Proof Let a program P be given, and let C be a clause in it.

We first determine the maximum number of times that an arbitrary pattern Δ can be evaluated against clause C, with $(pred\ C) = (pred\ \Delta)$. Such an evaluation is invoked due to adding the pair $\langle C, \Delta \rangle$ to $workset$ either initially or upon some iteration. By the biimplication (7) we have that if $\langle C, \Delta \rangle$ is added to $workset$ upon the n'th iteration, then for some body atom B_i of C we must have

$$
F^{n-1}(last\_call^n(C, \Delta, i)) \sqsubset F^n(last\_call^n(C, \Delta, i))
$$

and thus (by the invariant (5) and monotonicity of F^{n-1})

$$
F^{n-1}(last\_call^{n-1}(C, \Delta, i)) \sqsubset F^n(last\_call^n(C, \Delta, i)).
$$

For any body atom B_i of clause C, the size of the chain $\{\ldots, F^n(last\_call^n(C, \Delta, i)), \ldots\}$, with $0 \leq n \leq N$, is bounded by $DomHeight_C$. It follows that the pair $\langle C, \Delta \rangle$ is added to $workset$ initially plus at most $DomHeight_C *$ $(Atoms_C - 1)$ times.

Second, considering all patterns for clause C, and using that $DomHeight_C \geq 1$, we obtain that the total number of scans of C is bound by $DomSize_C * (1 + DomHeight_C * (Atoms_C - 1)) \leq Atoms_C * DomHeight_C * DomSize_C$.
□

Recall the definition of $Atoms_C$ etc. from the end of section 4. Letting

$$\begin{array}{lcl} DomSize_P & = & max_{C \in P}\ DomSize_C \\ DomHeight_P & = & max_{C \in P}\ DomHeight_C \\ Atoms_P & = & \Sigma_{C \in P}\ Atoms_C \end{array}$$

we obtain from the above proposition that the total number of clause scans during anaysis of an arbitrary program P is bound by $DomSize_P * DomHeight_P * Atoms_P$.

6 Discussion

An efficent algorithm for top-down or query directed analysis of logic programs has been defined in which the fixpoint iteration is controlled using a dynamic dependency relation.

The complexity result for the top-down algorithm is promising because the number of passes of dataflow analysis over each clause is independent of the size of the rest of the program. Considering all clauses in a given program, the total number of clause scans was shown to be bound by the following product: The maximum size of the domain of patterns for a predicate, times the height of that domain, times the number of atoms in the given program. Thus, the worst-case bound of [4] is improved by two orders of magnitude.

In the groundness analysis that has served as an example analysis, the number of patterns for a predicate is exponential in the arity of the predicate; and indeed, most practically useful analyses will make use of domains than even are larger than those considered here. To deal with this kind of combinatorial explosion, it may be useful to combine the application of a dynamic dependency relation with the use of widening and narrowing operators [8].

Another useful modification of the top-down algorithm is to incorporate a minimal function graph approach [14], as discussed in [12]. Execution of a minimal function graph (*mfg*) version of the algorithm begins with a set of patterns provided by the programmer, implying a constraint on how the program may be queried. Clauses are then only scanned for the externally given patterns plus those whose evaluation is required for the evaluation of the externally given ones. This approach may reduce the average cost of the algorithm. In addition, information may be provided about how predicates will be called during runtime, rather then merely about a programs input-output behavior as in the (non-*mfg*) abstract top-down semantics.

References

[1] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Logic Programming* 1987, **4**, pp. 259–262.

[2] F. Bancilhon and R. Ramakrishnan. *An Amateur's Intorduction to Recursive Query Processing Strategies.* MCC Technical Report No. DB-091-86, March 1986.

[3] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Logic Programming* 1991, **10**, pp. 225–299.

[4] 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, Paris, pp. 64–78.

[5] B. Le Charlier and P. Van Hentenryck. *Experimental evaluation of a generic abstract interpretation algorithm for Prolog*. Technical Report CS-91-55, August 1991, Brown University.

[6] 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, .

[7] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *Proc. ACM Symposium on Artificial Intelligence and programming languages*, SIGPLAN Notices, **12** (8),1977, pp. 1–12.

[8] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming* 1992 **13**, pp. 103–179.

[9] 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.

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

[11] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. An abstract machine for CLP($\mathcal{R}$). *SIGPLAN PLDI*, San Francisco, June 1992.

[12] N. Jørgensen. *Abstract interpretation of constraint logic programs.* Ph.D. thesis. To appear in *Datalogiske skrifter*, 1993, Computer Science Dept., Roskilde University Center.

[13] N. Jørgensen, K. Marriott, and S. Michaylov. Some global compile-time optimizations of CLP($\mathcal{R}$). *Proc. International Logic Programming Symposium*, San Diego, 1991, pp. 420–434.

[14] 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*, Florida, 1986, pp. 296–306.

[15] R.A. O'Keefe. Finite fixed-point problems. *Proc. 4th International Conference on Logic Programming*, 1987, Melbourne, 729–743.

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

[17] G.A. Kildall. A unified approach to global program optimization. *Proc. POPL 1973*, Boston, 194–206.

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

[19] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *J. Logic Programming* 1992, **13**, pp. 181–204.

[20] C. Martin and C. Hanking. Finding fixed points in finite lattices. *FPLCA 1987* Portland, Ohio.

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

[22] S. Peyton-Jones and C. Clack. Finding fixpoints in abstract interpretation, in: S. Abramski and C. Hankin (eds): *Abstract interpretation of declarative languages*. Ellis Horwood, 1987, pp. 246–265.

[23] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 1955, 5, pp. 285–309.

A Appendix

This appendix describes how the top-down algorithm proceeds when given program P_{ex} of example 4.1 as input.

Notation and execution order is as in example 4.3. If more than one pair is to be added to $workset$ upon an iteration, pair $\langle C, \Delta \rangle$ goes before $\langle C, \Delta' \rangle$ if Δ grounds an argument more to the left than the leftmost argument grounded by Δ'.

At initialization, the variables F, $depend$, and $workset$ are assigned values as shown in the top row of the table.

The algorithm performs a total of nine iterations. In the first iteration, the update of F in point $p(\mathsf{G}, \mathsf{G})$ leads to changes in F for the other patterns for p as well, due to preservation of monotonicity of the iterand. Beginning with the fifth iteration, the dependency relation is modified so as to take on non-trivial values. This is due to the evaluation of the recursive clause C_2. The seventh iteration leads to an update of F in point $p(\mathsf{A}, \mathsf{G})$ (and consequently in point $p(\mathsf{A}, \mathsf{A})$). The dependency relation $depends^7$ relates the pair $\langle C_2, p(\mathsf{G}, \mathsf{A}) \rangle$ to the update-point $p(\mathsf{A}, \mathsf{G})$, so this pair is inserted into $workset^7$. The algorithm terminates after the ninth iteration because then the $workset$ is empty.

n	Computation	F^n	$depend^n$	$workset^n$
0		$p(\mathsf{G},\mathsf{G}) \rightarrow \bot$ $p(\mathsf{G},\mathsf{A}) \rightarrow \bot$ $p(\mathsf{A},\mathsf{G}) \rightarrow \bot$ $p(\mathsf{A},\mathsf{A}) \rightarrow \bot$	$p(\mathsf{G},\mathsf{G}) \rightarrow \emptyset$ $p(\mathsf{G},\mathsf{A}) \rightarrow \emptyset$ $p(\mathsf{A},\mathsf{G}) \rightarrow \emptyset$ $p(\mathsf{A},\mathsf{A}) \rightarrow \emptyset$	$\{\ \langle C_1, p(\mathsf{G},\mathsf{G})\rangle, \langle C_1, p(\mathsf{G},\mathsf{A})\rangle,$ $\langle C_1, p(\mathsf{A},\mathsf{G})\rangle, \langle C_1, p(\mathsf{A},\mathsf{A})\rangle,$ $\langle C_2, p(\mathsf{G},\mathsf{G})\rangle, \langle C_2, p(\mathsf{G},\mathsf{A})\rangle,$ $\langle C_2, p(\mathsf{A},\mathsf{G})\rangle, \langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
1	$\mathbf{C}_{td}\,[\![C_1]\!]\,F^0\,p(\mathsf{G},\mathsf{G})$ $= p(\mathsf{G},\mathsf{G})$	$p(\mathsf{G},\mathsf{G}) \rightarrow p(\mathsf{G},\mathsf{G})$ $p(\mathsf{G},\mathsf{A}) \rightarrow p(\mathsf{G},\mathsf{G})$ $p(\mathsf{A},\mathsf{G}) \rightarrow p(\mathsf{G},\mathsf{G})$ $p(\mathsf{A},\mathsf{A}) \rightarrow p(\mathsf{G},\mathsf{G})$	$\vdots$	$\{\ \langle C_1, p(\mathsf{G},\mathsf{A})\rangle, \langle C_1, p(\mathsf{A},\mathsf{G})\rangle,$ $\langle C_1, p(\mathsf{A},\mathsf{A})\rangle, \langle C_2, p(\mathsf{G},\mathsf{G})\rangle,$ $\langle C_2, p(\mathsf{G},\mathsf{A})\rangle, \langle C_2, p(\mathsf{A},\mathsf{G})\rangle,$ $\langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
2	$\mathbf{C}_{td}\,[\![C_1]\!]\,F^1\,p(\mathsf{G},\mathsf{A})$ $= p(\mathsf{G},\mathsf{A})$	$p(\mathsf{G},\mathsf{G}) \rightarrow p(\mathsf{G},\mathsf{G})$ $p(\mathsf{G},\mathsf{A}) \rightarrow p(\mathsf{G},\mathsf{A})$ $p(\mathsf{A},\mathsf{G}) \rightarrow p(\mathsf{G},\mathsf{G})$ $p(\mathsf{A},\mathsf{A}) \rightarrow p(\mathsf{G},\mathsf{A})$	$\vdots$	$\{\ \langle C_1, p(\mathsf{A},\mathsf{G})\rangle, \langle C_1, p(\mathsf{A},\mathsf{A})\rangle,$ $\langle C_2, p(\mathsf{G},\mathsf{G})\rangle, \langle C_2, p(\mathsf{G},\mathsf{A})\rangle,$ $\langle C_2, p(\mathsf{A},\mathsf{G})\rangle, \langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
3	$\mathbf{C}_{td}\,[\![C_1]\!]\,F^2\,p(\mathsf{A},\mathsf{G})$ $= p(\mathsf{G},\mathsf{G})$	$\vdots$	$\vdots$	$\{\ \langle C_1, p(\mathsf{A},\mathsf{A})\rangle, \langle C_2, p(\mathsf{G},\mathsf{G})\rangle,$ $\langle C_2, p(\mathsf{G},\mathsf{A})\rangle, \langle C_2, p(\mathsf{A},\mathsf{G})\rangle,$ $\langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
4	$\mathbf{C}_{td}\,[\![C_1]\!]\,F^3\,p(\mathsf{A},\mathsf{A})$ $= p(\mathsf{G},\mathsf{A})$	$\vdots$	$\vdots$	$\{\ \langle C_2, p(\mathsf{G},\mathsf{G})\rangle, \langle C_2, p(\mathsf{G},\mathsf{A})\rangle,$ $\langle C_2, p(\mathsf{A},\mathsf{G})\rangle, \langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
5	$\mathbf{C}_{td}\,[\![C_2]\!]\,F^4\,p(\mathsf{G},\mathsf{G})$ $= p(\mathsf{G},\mathsf{G})$	$\vdots$	$p(\mathsf{G},\mathsf{G}) \rightarrow \{\langle C_2, p(\mathsf{G},\mathsf{G}), 1\rangle\}$ $p(\mathsf{G},\mathsf{A}) \rightarrow \emptyset$ $p(\mathsf{A},\mathsf{G}) \rightarrow \emptyset$ $p(\mathsf{A},\mathsf{A}) \rightarrow \emptyset$	$\{\ \langle C_2, p(\mathsf{G},\mathsf{A})\rangle, \langle C_2, p(\mathsf{A},\mathsf{G})\rangle,$ $\langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
6	$\mathbf{C}_{td}\,[\![C_2]\!]\,F^5\,p(\mathsf{G},\mathsf{A})$ $= p(\mathsf{G},\mathsf{G})$	$\vdots$	$p(\mathsf{G},\mathsf{G}) \rightarrow \{\langle C_2, p(\mathsf{G},\mathsf{G}), 1\rangle\}$ $p(\mathsf{G},\mathsf{A}) \rightarrow \{\langle C_2, p(\mathsf{A},\mathsf{G}), 1\rangle\}$ $p(\mathsf{A},\mathsf{G}) \rightarrow \emptyset$ $p(\mathsf{A},\mathsf{A}) \rightarrow \emptyset$	$\{\ \langle C_2, p(\mathsf{A},\mathsf{G})\rangle, \langle C_2, p(\mathsf{A},\mathsf{A})\rangle\ \}$
7	$\mathbf{C}_{td}\,[\![C_2]\!]\,F^6\,p(\mathsf{A},\mathsf{G})$ $= p(\mathsf{A},\mathsf{G})$	$p(\mathsf{G},\mathsf{G}) \rightarrow p(\mathsf{G},\mathsf{G})$ $p(\mathsf{G},\mathsf{A}) \rightarrow p(\mathsf{G},\mathsf{A})$ $p(\mathsf{A},\mathsf{G}) \rightarrow p(\mathsf{A},\mathsf{G})$ $p(\mathsf{A},\mathsf{A}) \rightarrow p(\mathsf{A},\mathsf{A})$	$p(\mathsf{G},\mathsf{G}) \rightarrow \{\langle C_2, p(\mathsf{G},\mathsf{G}), 1\rangle\}$ $p(\mathsf{G},\mathsf{A}) \rightarrow \{\langle C_2, p(\mathsf{A},\mathsf{G}), 1\rangle\}$ $p(\mathsf{A},\mathsf{G}) \rightarrow \{\langle C_2, p(\mathsf{G},\mathsf{A}), 1\rangle\}$ $p(\mathsf{A},\mathsf{A}) \rightarrow \emptyset$	$\{\ \langle C_2, p(\mathsf{A},\mathsf{A})\rangle, \langle C_2, p(\mathsf{G},\mathsf{A})\rangle\ \}$
8	$\mathbf{C}_{td}\,[\![C_2]\!]\,F^7\,p(\mathsf{A},\mathsf{A})$ $= p(\mathsf{A},\mathsf{A})$	$\vdots$	$p(\mathsf{G},\mathsf{G}) \rightarrow \{\langle C_2, p(\mathsf{G},\mathsf{G}), 1\rangle\}$ $p(\mathsf{G},\mathsf{A}) \rightarrow \{\langle C_2, p(\mathsf{A},\mathsf{G}), 1\rangle\}$ $p(\mathsf{A},\mathsf{G}) \rightarrow \{\langle C_2, p(\mathsf{G},\mathsf{A}), 1\rangle\}$ $p(\mathsf{A},\mathsf{A}) \rightarrow \{\langle C_2, p(\mathsf{A},\mathsf{A}), 1\rangle\}$	$\{\ \langle C_2, p(\mathsf{G},\mathsf{A})\rangle\ \}$
9	$\mathbf{C}_{td}\,[\![C_2]\!]\,F^8\,p(\mathsf{G},\mathsf{A})$ $= p(\mathsf{G},\mathsf{A})$	$\vdots$	$\vdots$	$\emptyset$