Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs

Javier Esparza
Technische Universität München
80290 München, Germany,
esparza@informatik.tu-muenchen.de

Andreas Podelski
Max-Planck-Institut für Informatik
66123 Saarbrücken, Germany
podelski@mpi-sb.mpg.de

Abstract

This paper is a contribution to the already existing series of work on the algorithmic principles of interprocedural analysis. We consider the generalization to the case of parallel programs. We give algorithms that compute the sets of backward resp. forward reachable configurations for parallel flow graph systems in linear time in the size of the graph viz. the program. These operations are important in dataflow analysis and in model checking. In our method, we first model configurations as terms (viz. trees) in the process algebra PA that can express call stack operations and parallelism. We then give a 'declarative' Horn-clause specification of the sets of predecessors resp. successors. The 'operational' computation of these sets is carried out using the Dowling-Gallier procedure for HornSat.

1 Introduction

The interprocedural dataflow analysis of sequential programs and the intraprocedural dataflow analysis of parallel programs have both been extensively studied (see for instance [15, 17] and the references therein). In this paper we go a step further, and study the *interprocedural* dataflow analysis of *parallel* programs. We do not impose any constraint on the interplay of procedures and parallelism. For instance, in the body of a procedure Π_1 both Π_1 and another procedure Π_2 can be called in parallel. This spins off a new instantiation of Π_2 each time Π_1 is called.

We model parallel programs with procedures as sets of parallel flow graphs (one for the main program and one for each procedure). Parallel flow graphs may contain procedure calls and **parbegin-parend** constructs.

In [10] the PA-algebra, a well-known process algebra [1], has been used to give these flow graphs a very simple operational semantics. The algebra contains two operators $\cdot$ and $\|$, which are used to express stack operations and parallelism. For example, the PA-term $\cdot(N_1, N_2)$— written $N_1 \cdot N_2$ in infix notation—models the configuration with control at program node n_1 that will, after the end of the current procedure, 'return' to n_2. The return itself is formally modelled by the fact that the term $N_1 \cdot N_2$ can be rewritten (possibly in many steps) to $\varepsilon \cdot N_2$, where ε is a special symbol modelling termination. The PA-term $(N_1 \| N_2) \cdot N_3$ models the configuration with control at program nodes n_1 and n_2 'in parallel' that will, after the corresponding **parend**, go to node n_3. The term $(N_1 \| N_2) \cdot N_3$ can be rewritten to $(\varepsilon \| \varepsilon) \cdot N_3$.

As shown in [10], many bitvector problems and other distributive data flow problems can be reduced to computing the sets $\mathrm{pre}(L), \mathrm{pre}^*(L), \mathrm{post}(L), \mathrm{post}^*(L)$ of immediate predecessors, predecessors, immediate successors and successors of certain *regular* sets L of PA-terms. A set of PA-terms is regular if the syntax trees of its elements form a regular tree language; a tree language is regular if it is accepted by a tree automaton (see [12]).

In a very interesting paper [18] (which, in fact, triggered this work), Lugiez and Schnoebelen prove that if a set L of PA-terms is regular then so are the sets $\mathrm{pre}(L), \mathrm{pre}^*(L), \mathrm{post}(L), \mathrm{post}^*(L)$ wrt. a given PA algebra Δ. In their complexity analysis, they focus on the number of states of the tree automata to be constructed, but not on the cost of a concrete algorithm for the construction itself (they only state that the construction can be implemented in polynomial time). They show that the number of states does not depend on the size of Δ (in fact, Δ can even be infinite). The constructions seem rather complicated, and they are derived *ad-hoc* for each of the two cases of pre* and of post*.

In this paper, we present simple algorithms, derived in a systematic way (see below), and we show that given a program of size n and a tree automaton of size m accepting a set of PA-terms L, our algorithms compute tree automata for $\mathrm{pre}(L), \mathrm{pre}^*(L), \mathrm{post}(L), \mathrm{post}^*(L)$ in $O(n \cdot m)$ time, i.e., in linear time in the size of the program if the size m of the tree automaton is assumed to be constant. Finally, our paper also contributes the application of the algorithms to some dataflow analysis problems.

In our approach we look at tree automata as a particularly simple class of logic programs. Our algorithm for the operation pre* (the algorithms for the other operations are similar) consists of a *declarative* and an *operational* step. In the declarative step, $\mathrm{pre}^*(L)$ is expressed as the least model of a logic program P_A which does *not* have the particularly simple form of tree automata, but can be directly (and easily) derived from the definition of pre*. In the operational step, P_A is transformed into an equivalent logic program that does correspond to a tree automaton; equivalence means here equality of the least models. The transformation is also performed in two stages. First, P_A undergoes a saturation procedure, after which all the clauses not corresponding to a tree automaton become redundant; then, these clauses are removed. The saturation procedure makes use of the Dowling-Gallier algorithm for HornSat [9]. The declarative and operational step run together in $O(n \cdot m)$ time, as mentioned above. This bound is a direct consequence of the fact that the Dowling-Gallier procedure runs in linear time. We thus avoid having to deal with worklist strategies and indexing techniques as is the case in other dynamic programming algorithms for similar problems.

Related Work. In the area of infinite-state model checking, a variety of systems performing push and pop operations on a single stack have been studied under the name of *context-free* and *pushdown processes*; for references see e.g. [3, 4, 6, 5, 11]). Model-checking techniques for context-free processes have inspired algorithms for dataflow analysis of sequential FGSs [16]. These algorithms are very different from ours, since they follow the classical approach of computing the "meet over all paths" semantics by means of the "maximal fixpoint semantics". In our approach we work directly with the "meet over all paths" semantics.

Reps [23] has also developed a non-classical approach to the interprocedural analysis of sequential programs based on algorithms for *CFL graph reachability.* Here, procedures are modelled by a restriction on valid paths (calls and returns must match, i.e. the edge labels must form a word in a context-free language). The problem can be solved by a dynamic programming algorithm which generalizes the CYK algorithm for CFL recognition and is related to the bottom-up evaluation of a special class of Datalog programs [23]. In the sequential case, the saturation part of our algorithm is reminiscent of Reps algorithm for CFL graph reachability, a connection that deserves further study.

The analysis of *parallel* programs as presented here can be contrasted with recent work by Ramalingam [22] that shows that synchronization-sensitive, context-sensitive interprocedural analysis of multi-tasking *concurrent* programs is undecidable. Evidently parallelism specified with parbegin-parend is less powerful than parallelism controlled by synchronication primitives.

Our algorithms are inspired by *set-based program analysis*, in which the abstract semantics of a program is the (generally) least solution of a *set constraint.* However, the logic program P_A mentioned above does not seem to correspond to any known class of set constraints (although other forms of logic programs do; see [6]).

Melski and Reps have shown that CFL graph reachability can be reduced to set constraint solving, and vice versa [20], and McAllester has shown that 'all' dynamic programming algorithms can be reduced to HornSat [19]. So the novelty of our contribution lies not so much in the general idea of applying set-based techniques and HornSat to the computation of $\mathrm{pre}(L), \mathrm{pre}^*(L), \mathrm{post}(L), \mathrm{post}^*(L)$, but in the concrete way of applying them.

Structure of the paper. The remainder of the paper is organized as follows. In Section 2 we introduce the flow graph model. The PA-algebra is introduced in Section 3, and a PA-semantics for the flow graph model is presented. Section 4 presents the algorithm for the predecessor operator pre*. Section 5 describes the changes needed to obtain the algorithms for the successor operator post*, the immediate-predecessor operator pre and the immediate-successor operator post. Section 6 sketches the application to dataflow analysis. Section 7 presents conclusions.

2 Parallel Flow Graphs

The *intra*procedural control flow of a single procedure is represented by a *flow graph* as in Figure 1. The nodes correspond to program points. The edges (expressing the control flow) are labeled by statements. Statements are assignments of the form $v := exp$ (where v is a variable, exp is an expression) or call statements of the form *call* $\Pi(Exp)$ (where Π is a procedure identifier, and Exp is a tuple of expressions). Control flow is interpreted nondeterministically; i.e., the guards of assignments are replaced by *true.* The *inter*procedural control flow of a sequential program with possibly several procedures is represented by a *flow graph system* (FGS) containing one flow graph for the main program and one flow graph for each procedure; see Figure 1.

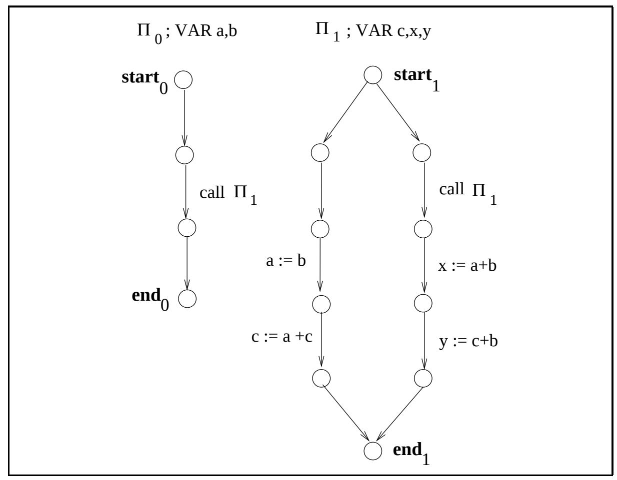

Figure 1: A flow graph system of a program with procedures.

In a *parallel FGS* we also allow hyperedges of the form $n \to \{n_1, \ldots, n_k\}$ that model a **parbegin** command, and hyperedges of the form $\{n_1, \ldots, n_k\} \to n$, modelling a **parend** command. Wlog. we assume $k = 2$. We assume that the **parbegin** and **parend** hyperedges are properly nested. We do not restrict the nesting of procedure calls and **parbegin-parend** instructions.

FGSs can be given a semantics in terms of *execution paths*, corresponding to the executions of the program with properly nested calls and returns. In the same manner (but with a much more complicated definition), parallel FGSs can be given a similar semantics in which parallelism is modelled by interleaving. We omit the formalization of this semantics; in the next section, we present a much simpler semantics using the PA-algebra. The PA-algebra semantics will clearly correspond to the expected behavior of a parallel FGS, and can be taken as fundamental semantics.

3 The Process Algebra PA

We introduce the syntax and semantics of the process algebra PA [1] closely following the presentation in [18].

Roughly, the process algebra specifies action-labeled transitions $t \xrightarrow{a} t'$ between states denoted by *PA-terms* t and t'. The term t' is obtained from t through rewriting of subterms.

The set T_{PA} of PA-terms is built up from finitely many given *process constants* and from the *empty process* ε using sequential composition "$\cdot$" and parallel composition "$\|$"; i.e. (we use t, t', t_1 etc. for PA-terms and X, Y, Z, X_1 etc. for process constants),

$$t ::= \varepsilon \mid X \mid t_1 \cdot t_2 \mid t_1 \| t_2 .$$

A *PA declaration* is a finite set Δ of *process rewrite rules* of the form $X \xrightarrow{a} t$, where X is a process constant, t is a PA-term and a is an *action* from a given finite set of *actions*.

Given a PA declaration Δ, the transition relation $\xrightarrow{a}$ over the set of PA-terms is the least relation satisfying the following inference rules, where the premises are placed above and the consequence below the horizontal line:

$$\mathbf{\Delta} \quad \frac{(X \xrightarrow{a} t) \in \Delta}{X \xrightarrow{a} t}$$

$$\mathbf{seq1} \quad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \cdot t_2 \xrightarrow{a} t_1' \cdot t_2}$$

$$\mathbf{seq2} \quad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \cdot t_2 \xrightarrow{a} t_1 \cdot t_2'} \quad (t_1 \in \mathrm{IsNil})$$

$$\mathbf{par1} \quad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \| t_2 \xrightarrow{a} t_1' \| t_2}$$

$$\mathbf{par2} \quad \frac{t_2 \xrightarrow{a} t_2'}{t_1 \| t_2 \xrightarrow{a} t_1 \| t_2'}$$

The rule **seq2** has an additional side condition (which can be seen as an additional premise). The set IsNil contains all PA-terms built up from the empty process with sequential and parallel composition. Intuitively, they correspond to the *terminated* terms, i.e. the terms that cannot execute any action. In particular **seq2** states that $t_1 \cdot t_2$ can do an a if t_1 is terminated and t_2 can do an a. If t_1 is not terminated then $t_1 \cdot t_2$ can do an a only if t_1 can.

The relation $\longrightarrow$ is the union of the relations $\xrightarrow{a}$ for all actions a. The *reachability relation* $\xrightarrow{*}$ is the reflexive and transitive closure of the relation $\longrightarrow$ that is the union of $\xrightarrow{a}$ for all actions a.

The set $\mathrm{pre}^*(L)$ of *predecessors* (with respect to the given PA declaration Δ) of a set of PA-terms L is the set of all PA-terms t that can reach a PA-term in L, i.e. $t \xrightarrow{*} t'$ for some $t' \in L$. The sets $\mathrm{pre}(L)$, $\mathrm{post}(L)$ and $\mathrm{post}^*(L)$ of *immediate predecessors*, *immediate successors*, and *successors* are defined similarly.

$$\begin{aligned}
\mathrm{pre}^*(L) &= \{t \mid t \xrightarrow{*} t' \text{ for some } t' \in L\} \\
\mathrm{pre}(L) &= \{t \mid t \longrightarrow t' \text{ for some } t' \in L\} \\
\mathrm{post}(L) &= \{t \mid t' \longrightarrow t \text{ for some } t' \in L\ \} \\
\mathrm{post}^*(L) &= \{t \mid t' \xrightarrow{*} t \text{ for some } t' \in L\}
\end{aligned}$$

The Translation of FGSs. We translate a parallel FGS into a PA declaration Δ. For each program node n we introduce a process constant N. The actions are the assignment statements of the program). The rewrite rules of Δ are as follows.

$$\begin{array}{ll}
N \longrightarrow M & \text{for } n \longrightarrow m \\[1ex]
N \xrightarrow{v := t} M & \text{for } n \xrightarrow{v := t} m \\[1ex]
N \longrightarrow START_i \cdot M & \text{for } n \xrightarrow{call\ \Pi_i(T)} m \\[1ex]
END_i \longrightarrow \varepsilon & \text{for end node of procedure } \Pi_i \\[1ex]
\begin{array}{l} N \longrightarrow (M_1 \| M_2) \cdot M, \\ M_1' \longrightarrow \varepsilon, \\ M_2' \longrightarrow \varepsilon \end{array} & \text{for} \quad \begin{array}{l} n \longrightarrow \{m_1, m_2\}, \\ \{m_1', m_2'\} \longrightarrow m. \end{array}
\end{array}$$

In this definition $n \xrightarrow{l} m$ means that the parallel FGS contains en edge between the program points n and m labeled by l. $N \longrightarrow M$ is an abbreviation of $N \xrightarrow{\tau} M$ for an special "silent" action τ. The definition assumes that the hyperedges $n \longrightarrow \{m_1, m_2\}$ and $\{m_1', m_2'\} \longrightarrow m$ match, i.e., that they correspond to a **parbegin-parend** instruction. For instance, the parallel flow graph for the program

$$\textbf{parbegin } x := 1, x := 2 \textbf{ parend}; \ y := x$$

is translated into the following PA declaration.

$$\begin{array}{rcl}
START & \longrightarrow & K \cdot N_3 \\
K & \longrightarrow & N_1 \| N_2 \\
N_1 & \xrightarrow{x:=1} & \varepsilon \\
N_2 & \xrightarrow{x:=1} & \varepsilon \\
N_3 & \xrightarrow{y:=x} & END \\
END & \longrightarrow & \varepsilon
\end{array}$$

A possible execution of the program is

$$\begin{aligned}
START &\longrightarrow K \cdot N_3 \\
&\longrightarrow (N_1 \| N_2) \cdot N_3 \\
&\xrightarrow{x:=2} (\varepsilon \| N_2) \cdot N_3 \\
&\xrightarrow{x:=1} (\varepsilon \| \varepsilon) \cdot N_3 \\
&\xrightarrow{y:=x} (\varepsilon \| \varepsilon) \cdot END \\
&\longrightarrow (\varepsilon \| \varepsilon) \cdot \varepsilon .
\end{aligned}$$

The terms in this execution describe the control of the program. For instance, the term $(N_1 \| N_2) \cdot N_3$ describes that control is at the nodes n_1 and n_2, and that after termination of the **parbegin-parend** instruction the execution will be resumed at n_3.

We now modify the translation of the two hyperedges $n \longrightarrow \{m_1, m_2\}$ and $\{m_1', m_2'\} \longrightarrow m$ as follows. We replace the rule $N \longrightarrow (M_1 \| M_2) \cdot M$ by two rules (using a new auxiliary symbol K). That is, the last case becomes:

$$\begin{array}{ll}
\begin{array}{l} N \longrightarrow K \cdot M, \\ K \longrightarrow M_1 \| M_2 \\ M_1' \longrightarrow \varepsilon, \\ M_2' \longrightarrow \varepsilon \end{array} & \text{for} \quad \begin{array}{l} n \longrightarrow \{m_1, m_2\}, \\ \{m_1', m_2'\} \longrightarrow m. \end{array}
\end{array}$$

Observe that now, the terms appearing in Δ are of depth 1, i.e. of the form ε, X, $X \cdot Y$ or $X \| Y$. This will play a role in the complexity analysis of the algorithm to be presented next.

4 The Algorithm for pre*

The problem is to compute the set $\text{pre}^*(L)$ of predecessors of a language L, wrt. a PA declaration Δ that is derived from a parallel FGS through the translation presented in the previous section. In particular, the terms appearing in Δ have depth 1.

The language L can be infinite; identifying a PA-term with its syntax tree, we only require L to be a *regular* set of trees, i.e. to have a finite representation in the form of a *tree automaton*. The set $\text{pre}^*(L)$ should also be represented as another tree automaton, incidentally proving that $\text{pre}^*(L)$ is regular whenever L is.

Following [6], we look at tree automata as a specially simple class of logic programs. We introduce this view of tree automata in Section 4.1.

We assume that L is a so-called ε-closed set of terms. Section 4.2 introduces these sets, and shows why this assumption can be made without loss of generality.

The algorithm consists of a *declarative* and an *operational* step. In the declarative step, $\text{pre}^*(L)$ is expressed as the least model of a logic program (denoted by P_A) which does *not* have the particularly simple form corresponding to a tree automaton, but can be directly (and easily) derived from the definition of $\text{pre}^*(L)$. This step is described in Section 4.3.

In the operational step P_A is transformed into an equivalent logic program $RedP_A$ that does correspond to a tree automaton, where equivalence means equality of the least models. This transformation is performed in two stages. First, a logic program $SatP_A$ is obtained from P_A by means of a *saturation* procedure. Then, some clauses are removed from $SatP_A$ to yield $RedP_A$. Both stages are described in Section 4.4.

An important point is that the operational part, which is the most complicated, is *the same* in the four cases $\text{pre}(L), \text{pre}^*(L), \text{post}(L), \text{post}^*(L)$. This allows to derive the four algorithms in an unified way (thus improving on the results of [18], where the two cases for predecessors and successors have to be considered separately). Only the formulation of the logic program P_A depends (in a rather straightforward way) on the particular case.

4.1 Tree Automata

In this paper (following the representation e.g. in [6, 5]), a tree automaton A is a special kind of logic program, namely a set of implications (Horn clauses) of the form:

$$q(f(x_1, \ldots, x_k)) \leftarrow q_1(x_1), \ldots, q_k(x_k)$$

where $k \geq 0$ is the arity of the function symbol f (if $k = 0$, we write $q(f)$ for the clause $q(f) \leftarrow \mathit{true}$; in our algorithm we will have $0 \leq k \leq 2$). We call Horn clauses of this form *reduction clauses.*

A tree t is *accepted* by A from state q if the atom $q(t)$ lies in the least model of A. We recall that this is equivalent to saying that the atom $q(t)$ is logically entailed by the program A (formally, $A \models q(t)$), or that the atom $q(t)$ has a successful derivation; since a derivation is isomorphic to a *run* of a *top-down* tree automaton as in [12, 21]), our notion of acceptance coincides with the standard one. The set of all trees accepted by a tree automaton A from state q is denoted by $L_q(A)$.[1] Thus,

$$L_q(A) = \{t \mid A \models q(t)\}.$$

If q is a fixed initial state of A, we write $L(A)$ for $L_q(A)$; if the language L is equal to $L(A)$, we say that L is *recognized* by A. Any set of trees L such that L is recognized by some tree automaton is called a *regular language.* An important property of tree automata is the fact that the tests of emptiness and of membership (for the recognized language) are linear [12].

Example of a Regular Language: IsNil**.** We recall that the set IsNil contains all PA-terms built up from the empty process and sequential and parallel composition. If we fix the predicate q_ε as the initial state, the set IsNil is recognized by the tree automaton given by the three clauses below.

$$\begin{array}{l} q_\varepsilon(\varepsilon) \\ q_\varepsilon(x_1 \cdot x_2) \leftarrow q_\varepsilon(x_1), q_\varepsilon(x_2) \\ q_\varepsilon(x_1 \| x_2) \leftarrow q_\varepsilon(x_1), q_\varepsilon(x_2) \end{array}$$

Example of a Regular Language: At_n**.** We define a regular set At_n of PA-terms which will be used later in Section 6. Intuitively, At_n is the set of PA-terms such that control is at the node n (and possibly also at some other nodes). Formally, we define that a node n is *active* at a PA-term t if

- $t = N$, or
- $t = t_1 \cdot t_2$ and n is active at t_1, or
- $t = t_1 \| t_2$ and n is active at t_1 or at t_2, or
- $t = t_1 \cdot t_2$ and $t_1 \in \text{IsNil}$ and n is active at t_2.

We denote by At_n the set of PA-terms at which n is active. This set is a regular language; it is recognized by the tree automaton below.

$$\begin{array}{r} q(N) \\ q(x_1 \cdot x_2) \leftarrow q(x_1) \\ q(x_1 \| x_2) \leftarrow q(x_1) \\ q(x_1 \| x_2) \leftarrow q(x_2) \\ q(x_1 \cdot x_2) \leftarrow q_\varepsilon(x_1), q(x_2) \end{array}$$

(all rules of the tree automaton for IsNil)

Note that this logic program is of *constant size* (i.e. not depending on the size of the flow graph). (In contrast, a tree automaton in the classical presentation [12] would amount to having clauses of the form $q(x_1 \cdot x_2) \leftarrow q(x_1), q_{all}(x_2)$ etc., where the predicate q_{all} stands for the state from which all terms are accepted. The definition of q_{all} requires a transition rule for each process constant.)

4.2 ε-Closure

A language L of PA-terms is ε*-closed* if the PA-terms $\varepsilon \cdot t$, $\varepsilon \| t$, and $t \| \varepsilon$ lie in L if and only if the PA-term t does.

We restrict the algorithm computing $\text{pre}^*(L)$ (or $\text{pre}(L)$, $\text{post}(L)$ or $\text{post}^*(L)$) to ε-closed languages L. This restriction is justified by the following facts (the first one relies crucially on the side condition for the structural rule **seq2**).

[1] We extend this notation to general logic programs P; thus, $L_q(P) = \{t \mid P \models q(t)\}$.

1. The PA-terms t, $\varepsilon \cdot t$, $t \| \varepsilon$ generate isomorphic transition sequences.[2]

2. If the language L is ε-closed then so are the languages $\text{pre}(L)$, $\text{pre}^*(L)$, $\text{post}(L)$ and $\text{post}^*(L)$.

3. If the language L is regular then so is its ε-closure.

By (1), the terms t, $\varepsilon \cdot t$, and $t\|\varepsilon$ are equivalent for all dataflow analysis purposes. The facts (2) and (3) guarantee that pre, pre*, post and post* are internal operations on regular ε-closed sets.

Every tree automaton recognizing a language L can be transformed into one recognizing the ε-closure of L. We only need to add a state q_ε and the clause $q_\varepsilon(\varepsilon)$ and, for every state q (including q_ε), the clauses

$$\begin{aligned} q(x_1 \cdot x_2) &\leftarrow q_\varepsilon(x_1), q(x_2), \\ q(x_1 \| x_2) &\leftarrow q_\varepsilon(x_1), q(x_2), \\ q(x_1 \| x_2) &\leftarrow q(x_1), q_\varepsilon(x_2). \end{aligned} \tag{1}$$

For any state q, the language L_q recognized by this new tree automaton from q is ε-closed. In particular, for $q = q_\varepsilon$, the recognized language is $L_{q_\varepsilon} = \text{IsNil}$. (Note that we could have defined IsNil as the ε-closure of the singleton set $\{\varepsilon\}$.)

4.3 The Declarative Part: Defining P_A

Given a PA declaration Δ and a tree automaton A accepting an ε-closed set L of PA-terms, we construct a logic program P_A with a distinguished predicate p_0 such that

$$t \in \text{pre}^*(L) \text{ iff } P_A \models p_0(t).$$

In other words, the PA-term t is a predecessor of a PA-term in L if and only if the atom $p_0(t)$ belongs to the least model of P_A.

We assume that the states of the tree automaton A are $q_0, q_1, \ldots, q_{n-1}, q_\varepsilon$ (we identify q_ε and q_n). We fix q_0 as the initial state, i.e. $L = L_{q_0}$. The automaton is given by a logic program consisting of reduction clauses of the form (where $0 \le i, j, k \le n$)

$$\begin{aligned} &q_i(\varepsilon) \quad \text{or} \\ &q_i(X) \quad \text{or} \\ &q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \quad \text{or} \\ &q_i(x \| y) \leftarrow q_j(x), q_k(y). \end{aligned}$$

We assume in particular that A contains reduction clauses of the form (1), according to the special role of the predicate q_ε.

We define P_A as the logic program consisting of all reduction rules of the tree automaton A and the additional clauses in Figure 2. These clauses define new predicates $p_0, p_1, \ldots, p_\varepsilon$. Schematically:

$$\boxed{\begin{aligned} P_A = \{&\text{clauses for } q_i\text{'s in } A\} \cup \\ \{&\text{clauses for } p_i\text{'s in Figure 2}\} \end{aligned}}$$

The intended meaning of the predicate p_i is that $p_i(t)$ can be derived from P_A if and only if $t \in \text{pre}^*(L_{q_i})$, or, loosely speaking, "$p_i = \text{pre}^*(q_i)$"; in particular, $p_0(t)$ can be derived from P_A iff $t \in \text{pre}^*(L)$.

[2] They are even strongly bisimilar.

$$\boxed{\begin{aligned}
&p_i(\mathcal{X}) \leftarrow q_i(\mathcal{X}) \\
&\qquad \text{for each } \mathcal{X} \in \{\text{process constants of } \Delta\} \cup \{\varepsilon\} \\
&p_i(X) \leftarrow p_i(t) \\
&\qquad \text{for each } \ X \xrightarrow{a} t \ \text{ in } \Delta \\
&p_i(x_1 \cdot x_2) \leftarrow p_j(x_1), q_k(x_2) \\
&\qquad \text{for each } \ q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \ \text{ in } A \\
&p_i(x_1 \cdot x_2) \leftarrow p_\varepsilon(x_1), p_i(x_2) \\
&\qquad \text{for each } \ q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \ \text{ in } A \\
&p_i(x_1 \| x_2) \leftarrow p_j(x_1), p_k(x_2) \\
&\qquad \text{for each } \ q_i(x \| y) \leftarrow q_j(x), q_k(y) \ \text{ in } A
\end{aligned}}$$

Figure 2: The clauses defining the predicates p_i in the logic program P_A for the successor operator pre*, wrt. the tree automaton A with states q_i (for $i = 0, \ldots, n$, where $q_n = q_\varepsilon$) and wrt. the PA declaration Δ.

As we did with q_ε and q_n, we identify p_ε and p_n. Observe that, by assumption, the tree automaton A contains the clauses

$$q_i(x \cdot y) \leftarrow q_\varepsilon(x), q_i(y)$$

for every $i = 0, \ldots, n$; thus, the program P_A contains the clauses

$$p_i(x_1 \cdot x_2) \leftarrow p_\varepsilon(x_1), p_i(x_2)$$

for every $i = 0, \ldots, n$. Since we identify q_ε and q_n, these clauses are a special case of the third kind of clauses defining the predicate p_i in Figure 2. We still list them in Figure 2 for systematic reasons.

From now on, we always use $\mathcal{X}$ as standing either for process constants or for the empty process ε.

In order to show that the intended meaning of p_i matches the real meaning, we first need the following characterization of the sets $\text{pre}^*(L_{q_i})$:

Proposition 1 The sets $\text{pre}^*(L_{q_i})$ (for $i = 0, 1, \ldots, n$) are the smallest sets such that the following holds:

1. if $\mathcal{X} \in L_{q_i}$, then $\mathcal{X} \in \text{pre}^*(L_{q_i})$;

2. if $X \xrightarrow{a} t$ is a rule in Δ and $t \in \text{pre}^*(L_{q_i})$, then $X \in \text{pre}^*(L_{q_i})$;

3. if $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ is a clause in A and $t_1 \in \text{pre}^*(L_{q_j})$ and $t_2 \in L_{q_k}$ then $t_1 \cdot t_2 \in \text{pre}^*(L_{q_i})$;

4. if $t_1 \in \text{pre}^*(\text{IsNil})$ and $t_2 \in \text{pre}^*(L_{q_i})$ then $t_1 \cdot t_2 \in \text{pre}^*(L_{q_i})$;

5. if $q_i(x_1 \| x_2) \leftarrow q_j(x_1), q_k(x_2)$ is a clause in A and $t_1 \in \text{pre}^*(L_{q_j})$ and $t_2 \in \text{pre}^*(L_{q_k})$ then $t_1 \| t_2 \in \text{pre}^*(L_{q_i})$.

Proposition 2 The sets $\text{pre}^*(L_{q_i})$ (for $i = 0, 1, \ldots, n$) are the smallest sets such that the following holds:

1. if $\mathcal{X} \in L_{q_i}$, then $\mathcal{X} \in \text{pre}^*(L_{q_i})$;

2. if $X \xrightarrow{a} t$ is a rule in Δ and $t \in \mathrm{pre}^*(L_{q_i})$, then $X \in \mathrm{pre}^*(L_{q_i})$;

3. if $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ is a clause in A and $t_1 \in \mathrm{pre}^*(L_{q_j})$ and $t_2 \in L_{q_k}$ then $t_1 \cdot t_2 \in \mathrm{pre}^*(L_{q_i})$;

4. if $t_1 \in \mathrm{pre}^*(\mathrm{IsNil})$ and $t_2 \in \mathrm{pre}^*(L_{q_i})$ then $t_1 \cdot t_2 \in \mathrm{pre}^*(L_{q_i})$;

5. if $q_i(x_1 \| x_2) \leftarrow q_j(x_1), q_k(x_2)$ is a clause in A and $t_1 \in \mathrm{pre}^*(L_{q_j})$ and $t_2 \in \mathrm{pre}^*(L_{q_k})$ then $t_1 \| t_2 \in \mathrm{pre}^*(L_{q_i})$.

Proof. We first prove that the sets $\mathrm{pre}^*(L_{q_i})$ satisfy the conditions, and then that they are the smallest such sets. Let us prove that the sets satisfy the third condition, the others being similar. Since $t_1 \in \mathrm{pre}^*(L_{q_j})$, there is a term $t_1' \in L_{q_j}$ such that $t_1 \xrightarrow{*} t_1'$. By repeated application of the rule **seq1** we have $t_1 \cdot t_2 \xrightarrow{*} t_1' \cdot t_2$. We prove $t_1' \cdot t_2 \in L_{q_i}$, which implies $t_1 \cdot t_2 \in \mathrm{pre}^*(L_{q_i})$. Since $t_1' \in L_{q_j}$ and $t_2 \in L_{q_k}$, we have $A \models q_j(t_1')$ and $A \models q_k(t_2)$. Since $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ is a clause of A, we also have $A \models q_i(t_1' \cdot t_2)$. So $t_1' \cdot t_2 \in L_{q_i}$.

To prove that $\mathrm{pre}^*(L_{q_i})$ are the smallest sets satisfying the properties specified in Conditions 1 to 5, let $S_0, \ldots, S_n$ be arbitrary sets satisfying the properties (i.e., Conditions 1 to 5 hold if we replace if we replace $\mathrm{pre}^*(L_{q_i})$ by S_i). We prove that for every term t and for every $i = 0, \ldots, n$, if $t \in \mathrm{pre}^*(L_{q_i})$ then $t \in S_i$.

We write $t \xrightarrow{k} t'$ to abbreviate that there is a sequence of rewriting steps from t to t' whose length is smaller than or equal to k. Thus, $t \in \mathrm{pre}^*(L_{q_i})$ means that $t \xrightarrow{*} t'$ for some $t' \in L_{q_i}$. We proceed by induction on k to prove the following statement:

$$\text{for all } k \;\; \text{for all } t \;\; \text{for all } i \;\; (\text{if } t \xrightarrow{k} t' \in L_{q_i} \text{ then } t \in S_i).$$

Base Case: $k = 0$. We proceed by structural induction on t to show:

$$\text{for all } t \;\; \text{for all } i \;\; (\text{if } t \in L_{q_i} \text{ then } t \in S_i).$$

- $t = \mathcal{X} \in L_{q_i}$. By Condition 1, $t \in S_i$.
- $t = t_1 \cdot t_2 \in L_{q_i}$. There exists a clause $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ in A such that $t_1 \in L_{q_j}$ and $t_2 \in L_{q_k}$. By induction hypothesis (of the induction on the structure of t), $t_1 \in S_j$. By Condition 3, $t \in S_i$. (Condition 4 is here the special case of Condition 3 for $j = n$.)
- $t = t_1 \parallel t_2 \in L_{q_i}$. This case follows the lines of the previous case, using Condition 5.

Induction Step: $k > 0$. We proceed by structural induction on t to show:

$$\text{for all } t \;\; \text{for all } i \;\; (\text{if } t \xrightarrow{k} t' \in L_{q_i} \text{ then } t \in S_i).$$

- $t = X \in L_{q_i}$. We assume $X \longrightarrow t'' \xrightarrow{k-1} t' \in L_{q_i}$ for some term t_2''. Then, by induction hypothesis (of the induction on k), we have $t'' \in S_i$. From $X \rightarrow t''$ we infer $X \xrightarrow{a} t'' \in \Delta$ for some action a. By Condition 2, $X \in S_i$.
- $t = t_1 \cdot t_2 \in L_{q_i}$. A simple inspection of the operational semantics of the PA-algebra shows that there are two possible cases:
  - $t'' = t_1'' \cdot t_2$ and $t_1 \xrightarrow{k} t_1'$ ("the rewriting in the left subterm is non-terminating"). Since $t' \in L_{q_i}$ there is a clause $q_i(x_1, x_2) \leftarrow q_j(x_1), q_k(x_2)$ in A such that $t_1' \in L_{q_j}$ and $t_2 \in L_{q_k}$. We infer $t_1 \in S_j$ from $t_1 \xrightarrow{k} t_1' \in L_{q_j}$ by induction hypothesis (of the induction on t). By Condition 3, $t = t_1 \cdot t_2 \in S_i$.
  - $t' = t_1' \cdot t_2'$ and $t_1' \in \mathrm{IsNil}$ and $t_1 \xrightarrow{k} t_1'$ and $t_2 \xrightarrow{k} t_2'$ ("the rewriting in the left subterm is terminating"). Since $t_1 \xrightarrow{*} t_1' \in \mathrm{IsNil}$, we have $t_1 \in \mathrm{pre}^*(\mathrm{IsNil})$. Since $t_1' \cdot t_2' \in L_{q_i}$ and $t_1' \in \mathrm{IsNil}$, we have $t_2' \in L_{q_i}$; here we use our assumption that A contains reduction clauses of the form (1) for every state q. That is, for some decomposition of k into $k = k' + k''$, we have[3]

    $$t_1 \xrightarrow{k'} t_1' \xrightarrow{k''} t_1' \in \mathrm{IsNil}$$
    $$t_2 \xrightarrow{k'} t_2 \xrightarrow{k''} t_2' \in L_{q_i}.$$

    We apply the induction hypothesis (of the induction on t) on the fact $t_2 \in \mathrm{pre}^*(L_{q_i})$ (which holds because $t_2 \xrightarrow{k} t_2' \in L_{q_i}$) and obtain $t_2 \in S_i$. By Condition 4, $t = t_1 \cdot t_2 \in S_i$.
- $t = t_1 \parallel t_2 \in L_{q_i}$. This case is very similar to the first subcase of the case for $t = t_1 \cdot t_2$. Here, we use Condition 5 instead of Condition 3 to show $t \in S_i$. ☐

We can now prove that the intended meaning of the predicates p_i coincides with its formal meaning.

Theorem 1 ("$\mathrm{pre}^*(q_i) = p_i$") A PA-term t is a predecessor of some PA-term in the language L_{q_i} recognized by A from the state q_i if and only if $p_i(t)$ lies in the least model of the logic program P_A. Formally,

$$\mathrm{pre}^*(L_{q_i}) = \{t \in T_{\mathrm{PA}} \mid P_A \models p_i(t)\}.$$

Proof. The rules of P_A model exactly the conditions defining the sets $\mathrm{pre}^*(L_{q_i})$ in Proposition 2. Thus, the sets $\{t \in T_{\mathrm{PA}} \mid P_A \models p_i(t)\}$ are the smallest sets satisfying those conditions. Now, we only need to apply Proposition 2 in order to obtain the statement. ☐

4.4 The Operational Part: $P_A \mapsto SatP_A \mapsto RedP_A$

In the first stage of the operational part of the algorithm, we *saturate* P_A. This means that we infer all clauses of the form $p(\mathcal{X})$ (where $\mathcal{X}$ is a process constant X of Δ or the empty process ε) such that $P_A \models p(\mathcal{X})$, and add them to P_A. The result is the *saturated* logic program $SatP_A$. Observe that the added clauses are a special case of reduction clauses. Schematically:

$$\boxed{SatP_A = P_A \cup \{p(\mathcal{X}) \mid P_A \models p(\mathcal{X})\}}$$

[3] According to the semantics of the PA algebra, the steps rewriting the left subterm precede the steps rewriting the right subterm. However, the proof that the term $t = t_1 \cdot t_2$ is a predecessor of some term in L_{q_i} proceeds by two proofs applied to the two subterms in any order (one deriving that the left subterm is a predecessor of a term in IsNil and one deriving that the right subterm is a predecessor of a term in L_{q_i}). This is perhaps the intuitive explanation for the efficiency of the algorithm.

In the second stage, we define $RedP_A$ as the logic program consisting of all reduction clauses of $SatP_A$, schematically:

$$\boxed{RedP_A = \{\text{reduction clauses in } SatP_A\}}$$

We show that all clauses in $SatP_A$ that are not reduction clauses are redundant in $SatP_A$, in the sense that omitting them does not change the least model.

The logic program $RedP_A$ is a tree automaton. Having fixed q_0 as the initial state of A, we fix p_0 as the initial state for the tree automaton $RedP_A$. This tree automaton, which recognizes the set $\mathrm{pre}^*(L)$, is the output of the algorithm.

Saturating P_A via HornSat. The only clauses in P_A that contain variables are reduction clauses of the form

$$r(x_1 \circ x_2) \leftarrow r_1(x_1), r_2(x_2)$$

where r, r_1 and r_2 are q_i or p_i for some i between 0 and n and "$\circ$" is either "$\cdot$" or "$\|$". If the number of clauses in A is m_A, then the number of clauses of P_A containing variables is $2\,m_A$ (P_A contains one new clause defining p_i for each 'old' clause defining q_i; see Figure 2).

We now define the logic program P_A^{ground} as the result of replacing each clause with variables, which is necessarily of the form $r(x_1 \circ x_2) \leftarrow r_1(x_1), r_2(x_2)$, by a set of ground clauses. This set contains a clause $r(t_1 \circ t_2) \leftarrow r_1(t_1), r_2(t_2)$ for each rewriting rule $X \xrightarrow{a} t_1 \circ t_2$ in Δ. If n_Δ is the size of the PA declaration measured by its number of rules (viz. the size of the parallel flow graph measured by its number of edges), then the number of all such instantiations is $O(m_A \cdot n_\Delta)$. Thus, the total size of P_A^{ground} is bounded by $O(m_A \cdot n_\Delta)$.

The interest of the logic program P_A^{ground} lies in the following proposition.

Proposition 3 If a clause $r(\mathcal{X})$ is a consequence of P_A then also of P_A^{ground}, formally

$$P_A \models r(\mathcal{X}) \;\text{ if and only if }\; P_A^{\text{ground}} \models r(\mathcal{X}).$$

Proof. The "if" direction is trivial. For the other direction, one can show, by induction over the length of a derivation for $r(\mathcal{X})$ wrt. the logic program P_A, that each atom in the derivation is a ground atom $p(t)$ (where t is a PA-term of depth at most 1) such that, if $p(t)$ unifies with the head of a clause in P_A then P_A^{ground} contains a ground instance of that clause whose head is $p(t)$; i.e., each resolution step wrt. P_A is possible also wrt. P_A^{ground}. $\square$

Proposition 3 reduces the problem of saturating P_A to the problem of saturating P_A^{ground} and then deriving all consequences of the form $r(\mathcal{X})$ from the set P_A^{ground} of *ground* Horn clauses. This is an instance of HornSat, where the propositional constants are the atoms of the form $r(\mathcal{X})$ and $r(X \circ Y)$ appearing in P_A^{ground}. This problem can be solved in linear time by the Dowling-Gallier procedure [9]. More precisely, this procedure computes the set of all derivable atoms in linear time (in the size of the logic program). (The idea of the Dowling-Gallier procedure is to iterate the following instruction: for each propositional constant forming the head of a clause with an empty body, remove the propositional constant from all clauses where it appears in the body.) Therefore, since the size of $SatP_A$ is $O(m_A \cdot n_\Delta)$, the program $SatP_A$ is obtained from the program P_A in $O(m_A \cdot n_\Delta)$ time.

From $SatP_A$ to $RedP_A$. In the second stage of the operational part, we obtain the logic program $RedP_A$ by removing from $SatP_A$ all non-reduction clauses. The output of the algorithm is the program $RedP_A$ (which consists of reduction rules only) as a tree automaton representing the set $\mathrm{pre}^*(L)$ of all predecessors of L wrt. Δ.

We show that $SatP_A$ and $RedP_A$ are equivalent, i.e., that the non-reduction clauses of $SatP_A$ are redundant.

Proposition 4 *$SatP_A$ is equivalent to $RedP_A$, i.e., the following set is empty:*

$$M = \{r(t) \mid SatP_A \models r(t) \text{ and } RedP_A \not\models r(t)\}.$$

Proof. For a proof by contradiction, assume that M is not empty. Let $r(t)$ be an element of M with the shortest derivation wrt. the logic program $SatP_A$. This derivation must have an application of a clause which is not in $RedP_A$. Such a clause is of the form $p_i(\mathcal{X}) \leftarrow \ldots$. This means that we have found an atom $p_i(\mathcal{X})$ (the one to which this clause is applied) that has a derivation wrt. $SatP_A$ (a derivation using a the clause $p_i(\mathcal{X}) \leftarrow \ldots$ which is not a reduction clause). Thus, the atom $p_i(\mathcal{X})$ lies in the least model of $SatP_A$, which is equal to the least model of $P_A \models p_i(\mathcal{X})$. By the construction of $SatP_A$ and by Proposition 3, $SatP_A$ contains all of its consequences in the form of a ground atom, and in particular the clause $p_i(\mathcal{X})$. It follows that the derivation of $r(t)$ wrt. the logic program $SatP_A$ can be made at least one step shorter, namely by applying the clause $p_i(\mathcal{X})$. This is a contradiction. $\square$

Correctness. Since $RedP_A$ is the output of the algorithm, correctness is stated as follows.

Theorem 2 Given the PA declaration Δ, a PA-term t is a predecessor of a PA-term in the language L_{q_i} recognized by the tree automaton A from the state q_i if and only if t is recognized by the tree automaton $RedP_A$ from state p_i. Formally,

$$\mathrm{pre}^*(L_{q_i}) = \{t \in T_{\mathrm{PA}} \mid RedP_A \models p_i(t)\}.$$

In particular, $\mathrm{pre}^*(L)$ is the set of PA-terms recognized by $RedP_A$.

Proof. By Proposition 3 and Proposition 4, $P_A \models p_i(t)$ iff $RedP_A \models p_i(t)$. Apply now Theorem 1.

The last statement of the theorem is the instance for $i = 0$, since we fixed q_0 as the initial state of A and p_0 as the initial state of $RedP_A$ (i.e. $L = L_{q_n}(A)$ and $\mathrm{pre}^*(L) = L_{p_n}(RedP_A)$). $\square$

Complexity. Since P_A^{ground} can be constructed in $O(m_A \cdot n_\Delta)$ time, the complete algorithm runs in $O(m_A \cdot n_\Delta)$ time; i.e., the algorithm is linear in the size m_A of the tree automaton A (i.e. the number of its clauses) and linear in the size n_Δ of the PA declaration Δ (i.e. the number of its rewrites rules, which is the number of edges of the parallel flow graph). In many applications to data flow analysis, the tree automaton A can be viewed as a constant parameter in the problem formulation with PA algebras (see Section 6), i.e., m_A can be assumed constant.

The number of states of the computed tree automaton representing $\mathrm{pre}^*(L)$ is twice the number k_A of states of the automaton representing the language L (to compare, the tree automaton obtained in [18] has $4\,k_A$ states); i.e., this bound does not depend on the input PA declaration. In contrast, the number of its clauses is bounded by $2\,m_A + k_\Delta \cdot k_A$ where k_Δ is the number of process constants of Δ.

The algorithms for $\mathrm{pre}(L)$, $\mathrm{post}^*(L)$ and $\mathrm{post}(L)$ are obtained in a similar way (see below) and have the same complexity.

$\mathrm{pre}^*(C)$	(C is set of configurations, pre^* refers to flow graph system)
$= \mathrm{pre}^*(L)$	(L is ε-closure of set of PA-terms, represented by tree automaton A, pre^* refers to PA declaration Δ)
$= lm(P_A)$	(P_A is a logic program (Figure 2))
$= lm(SatP_A)$	(saturation via HornSat)
$= lm(RedP_A)$	(reduction clauses of $SatP_A$})

Figure 3: Schematically, the steps of the algorithm for pre^*; the PA declaration Δ is obtained by translating a flow graph system (Section 3); the notation lm stands for 'least model'.

5 The Algorithms for post^*, pre and post

We only need to specify the declarative part of the algorithms computing $\mathrm{post}^*(L)$, $\mathrm{pre}(L)$ and $\mathrm{post}(L)$, respectively, for a set L of PA-terms given by the tree automaton A, wrt. a given PA algebra Δ; the operational part is the same as for pre^*.

The Algorithm for post^***.** We assume the setting described in Section 4.3, where we replace pre^* by post^*. The analogue of Proposition 2 is the following statement.

Proposition 5 *The sets* $\mathrm{post}^*(L_{q_i})$ *(for* $i = 0, 1, \ldots, n$*) are the smallest sets such that the following holds:*

1. *if* $\mathcal{X} \in L_{q_i}$ *then* $\mathcal{X} \in \mathrm{post}^*(L_{q_i})$*;*
2. *if* $X \xrightarrow{a} t$ *is a rule in* Δ *and* $X \in \mathrm{post}^*(L_{q_i})$*, then* $t \in \mathrm{post}^*(L_{q_i})$*;*
3. *if* $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ *is a clause in* A *and* $t_1 \in \mathrm{post}^*(L_j)$ *and* $t_2 \in L_k$ *then* $t_1 \cdot t_2 \in \mathrm{post}^*(L_{q_i})$*;*
4. *if* $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ *is a clause in* A *and* $t_1 \in \mathrm{post}^*(L_j) \cap \mathrm{IsNil}$ *and* $t_2 \in \mathrm{post}^*(L_k)$ *then* $t_1 \cdot t_2 \in \mathrm{post}^*(L_{q_i})$*;*
5. *if* $q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2)$ *is a clause in* A *and* $t_1 \in \mathrm{post}^*(L_j)$ *and* $t_2 \in \mathrm{post}^*(L_k)$ *then* $t_1 \| t_2 \in \mathrm{post}^*(L_{q_i})$*.*

Proof. The proof is analogous to the one of Proposition 2. □

We define $P_A^{\mathrm{post}^*}$ as the logic program that consists of all reduction rules of the tree automaton A and the additional clauses in Figure 4. (Recall that $\mathrm{IsNil} = L_{q_\varepsilon}$.) Schematically:

$$\boxed{\begin{aligned} P_A^{\mathrm{post}^*} = &\{\text{clauses for } q_i\text{'s in } A\} \cup \\ &\{\text{clauses for } p_i\text{'s in Figure 4}\}\end{aligned}}$$

The meaning of the new predicates p_i defined by the clauses in Figure 4 is that "$p_i = \mathrm{post}^*(q_i)$", i.e., that the atom $p_i(t)$ lies in the least model of $P_A^{\mathrm{post}^*}$ if and only if the PA-term t is reachable from a PA-term in L_{q_i}, formally

$$L_{p_i}(P_A^{\mathrm{post}^*}) = \mathrm{post}^*(L_{q_i}(A)).$$

$$p_i(\mathcal{X}) \leftarrow q_i(\mathcal{X})$$
for each $\mathcal{X} \in \{$process constants of $\Delta\} \cup \{\varepsilon\}$

$$p_i(t) \leftarrow p_i(X)$$
for each $X \xrightarrow{a} t$ in Δ

$$p_i(x_1 \cdot x_2) \leftarrow p_j(x_1), q_k(x_2)$$
for each $q_i(x \cdot y) \leftarrow q_j(x), q_k(y)$ in A

$$p_i(x_1 \cdot x_2) \leftarrow p_j(x_1), q_\varepsilon(x_1), p(x_2)$$
for each $q_i(x \cdot y) \leftarrow q_j(x), q_k(y)$ in A

$$p_i(x_1 \| x_2) \leftarrow p_j(x_1), p_k(x_2)$$
for each $q_i(x \| y) \leftarrow q_j(x), q_k(y)$ in A

Figure 4: The clauses defining the predicates p_i in the logic program $P_A^{\mathrm{post}^*}$ for the successor operator post^*, wrt. the tree automaton A with states q_i (for $i = 0, \ldots, n$, where $q_n = q_\varepsilon$) and wrt. the PA declaration Δ.

We note that the fourth kind of clauses in Figure 4 is a special form of a reduction clause containing two atoms with the same variable. The saturation procedure will lead to an *alternating* tree automaton, i.e. one that contains reduction clauses with conjunctions of atoms with the same variable. The membership test for alternating tree automata is still linear, while the emptiness test is exponential in general. In this case, however, we can replace the conjunction

$$p_j(x_1), q_\varepsilon(x_1)$$

by the atom $p_j^\varepsilon(x_1)$ and add the following clauses defining the new predicates p_i^ε, for $i = 0, \ldots, n$.

$$\begin{aligned} p_i^\varepsilon(\varepsilon) &\leftarrow p_i(\varepsilon) \\ p_i^\varepsilon(x_1 \cdot x_2) &\leftarrow q_\varepsilon(x_1), p_i^\varepsilon(x_2) \\ p_i^\varepsilon(x_1 \| x_2) &\leftarrow q_\varepsilon(x_1), p_i^\varepsilon(x_2) \\ p_i^\varepsilon(x_1 \| x_2) &\leftarrow p_i^\varepsilon(x_1), q_\varepsilon(x_2) \end{aligned}$$

The meaning of p_i^ε is given by

$$L_{p_i^\varepsilon}(P_A^{\mathrm{post}^*}) = \mathrm{post}^*(L_{q_i}(A)) \cap \mathrm{IsNil}.$$

The Algorithms for pre **and** post**.** We omit the analogue of Proposition 2 or 5, respectively, and instead give directly the clauses that need to be added to the reduction clauses of the tree automaton A in order to define the logic programs P_A^{pre} for the immediate-predecessor operator and P_A^{post} for the immediate-successor operator. We obtain these clauses by small changes of Figure 2 and 4, respectively. Namely, we omit the first kind of clauses ("the immediate-predecessor and immediate-successor relations are not reflexive") and replace the predicate p_i by the predicate q_i in the second kind of clauses ("the immediate-predecessor and immediate-successor relations are not transitive"); the other clauses remain the same.

$$
\begin{array}{ll}
p_i(X) \leftarrow q_i(t) & \\
& \text{for each } \; X \xrightarrow{a} t \; \text{ in } \Delta \\[1em]
p_i(x_1 \cdot x_2) \leftarrow p_j(x_1), q_k(x_2) & \\
& \text{for each } \; q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \; \text{ in } A \\[1em]
p_i(x_1 \cdot x_2) \leftarrow p_\varepsilon(x_1), p_i(x_2) & \\
& \text{for each } \; q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \; \text{ in } A \\[1em]
p_i(x_1 \| x_2) \leftarrow p_j(x_1), p_k(x_2) & \\
& \text{for each } \; q_i(x \| y) \leftarrow q_j(x), q_k(y) \; \text{ in } A
\end{array}
$$

Figure 5: The clauses defining the predicates p_i in the logic program P_A^{pre} for the immediate-predecessor operator, wrt. the tree automaton A with states q_i (for $i = 0, \ldots, n$, where $q_n = q_\varepsilon$), and wrt. the PA declaration Δ.

$$
\begin{array}{ll}
p_i(t) \leftarrow q_i(X) & \\
& \text{for each } \; X \xrightarrow{a} t \; \text{ in } \Delta \\[1em]
p_i(x_1 \cdot x_2) \leftarrow p_j(x_1), q_k(x_2) & \\
& \text{for each } \; q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \; \text{ in } A \\[1em]
p_i(x_1 \cdot x_2) \leftarrow p_j(x_1), q_\varepsilon(x_1), p(x_2) & \\
& \text{for each } \; q_i(x \cdot y) \leftarrow q_j(x), q_k(y) \; \text{ in } A \\[1em]
p_i(x_1 \| x_2) \leftarrow p_j(x_1), p_k(x_2) & \\
& \text{for each } \; q_i(x \| y) \leftarrow q_j(x), q_k(y) \; \text{ in } A
\end{array}
$$

Figure 6: The clauses defining the predicates p_i in the logic program P_A^{post} for the immediate-successor operator, wrt. the tree automaton A with states q_i (for $i = 0, \ldots, n$, where $q_n = q_\varepsilon$) and wrt. the PA declaration Δ.

6 Applications

Our algorithms can be used to solve bitvector problems and other distributive dataflow analysis problems for parallel FGSs along the lines of [10]. We briefly sketch the solution of [10] to a simple problem, namely whether a global variable v is *live* at a program point n. Then we show how to extend the technique to the case in which the program has both local and global variables, a problem that was left open in [10].

Assume the original parallel FGS has been translated into a PA declaration Δ. As usual, a program variable v is said to be *live* at the program point corresponding to the node n of the FGS if there is program path starting at (a program state with control at) n in which v is referenced before being redefined. (For simplicity, we assume that no instruction simultaneously redefines and references a variable.) If $START$ denotes the process constant corresponding to the start node of the main program, then this is equivalent to saying that there exists a rewriting sequence of PA-terms wrt. the PA declaration Δ,

$$START \xrightarrow{\sigma_1} t_1 \xrightarrow{\sigma_2} t_2 \xrightarrow{u := exp(v)} t_3$$

where the PA-term t_1 is reached from the PA-term $START$, and the PA-term t_2 reached from the PA-term t_1 after steps with actions that form the string σ, such that the following properties are satisfied.

1. The PA-term $START$ can reach a PA-term t_1 such that node n is *active* at t_1, formally $START \longrightarrow t_1 \in \mathrm{At}_n$ ("the program execution reaches a state with control at n").
2. No action of the form $v := exp$ is contained in the string σ ("the variable v is not defined during the execution of the statements in σ after that state with control at n").
3. The PA-term t_2 can be rewritten using a rule with an action of the form $u := exp(v)$ that stands for an assignment of an expression containing the variable v ("the variable v is referenced").

Let Δ_r be the subset of rewrite rules of Δ of the form

$$X \xrightarrow{u := exp} t$$

where the variable v appears in the expression exp. Then, the set of terms t_2 satisfying the property described under (3) is $\mathrm{pre}_{\Delta_r}(T_{PA})$, where the immediate-predecessor operator pre_{Δ_r} refers to the reachability relation of Δ_r.

Let Δ_{nd} be the subset of rewrite rules of Δ obtained by removing all rules of the form

$$X \xrightarrow{v := exp} t$$

from Δ; the predecessor operator $\mathrm{pre}^*_{\Delta_{nd}}$ refers to the reachability relation of Δ_{nd}. Then, the set of terms t_1 such that $t_1 \xrightarrow{\sigma} t_2$ for some string σ satisfying the property described under (2) is $\mathrm{pre}^*_{\Delta_{nd}}(\mathrm{pre}_{\Delta_r}(T_{PA}))$.

Finally, the subset of the terms t_1 satisfying the property described under (1) is

$$\mathrm{At}_n \cap \mathrm{post}^*_\Delta(\{START\}) \cap \mathrm{pre}^*_{\Delta_{nd}}(\mathrm{pre}_{\Delta_r}(T_{PA})).$$

This set of PA-terms can be computed using the results of Section 4 and standard algorithms for computing the intersection of regular tree languages. (It is possible to add some optimizations which are out of the scope of this paper.)

In the same manner we can solve the main bitvector problems of Hecht's hierarchy [13], namely the computation of very busy expressions, available expressions, and reaching definitions. In order to deal with kills in independent threads, intersection problems such as available expressions require that the problem is put in dual form (i.e., one solves the union problem of unavailable expressions and then complements the answer).

Local variables. When both local and global variables are present, it is necessary to distinguish between different incarnations of the same variable. This can be achieved by translating the program into a new PA declaration Δ'. The intuition is that Δ' simulates a new program that could be obtained by a source-to-source translation of the old one such that the new program contains a 'copy' of each procedure; the copy can be called (nondeterministically) at most once in any execution path (at any point from where the original procedure can be called); we analyze, say, the liveness of a local variable (which is based on the *existence* of an execution path) wrt. the copy. Thus, the PA declaration Δ' has the same rewrite sequences of PA-terms as Δ except for the fact that, for each procedure Π_i, at most one call is *marked*. Formally, this means that a rewriting rule

$$N \xrightarrow{mark} START_i \cdot M$$

with the special action symbol *mark* is applied at most once for each i (where $START_i$ is the process constant corresponding to the start node of the procedure Π). Termination of the marked call is signaled by executing an action *return*.

The restriction to at most one application to the rewriting rule above (translating a marked procedure call) is obtained by 'coloring' the process constants. That is, each process constant X of Δ is split in Δ' into three process constants X^g, X^r, X^b, where the superscripts stand for green, red, and black. The rules of Δ' are defined in such a way that

(a) red process constants can only be generated at the marked call;

(b) green process constants can only be generated in the same computation thread as the marked call, and before the marked call starts, i,e, before the process constant *mark*;

(c) black process constants can only be generated in parallel to the marked call (i.e., in computation threads parallel to the one which initiated the marked call), or after the marked call has terminated;

(d) every run contains the process constant *mark* at most once.

For instance, a rule $N \longrightarrow START_i \cdot M$ of Δ is replaced in Δ' by the following set of rules:

- $N^b \longrightarrow START_i^b \cdot M^b$
  Black constants only generate black constants.
- $N^g \xrightarrow{mark} START_i^r \cdot M^b$.
  The procedure call is marked and $START_i$ becomes red.
- $N^g \longrightarrow START_i^g \cdot M^b$
  This rule does not mark the call of the procedure Π_i, and by coloring M black it guesses that some call will be marked before execution is resumed at M (notice that this is only a guess, because the green constant $START_i^g$ may, but does not necessarily have to, generate a red constant). The guess may be wrong, but in this case the run contains no marked call at all, which is harmless.
- $N^g \longrightarrow START_i^b \cdot M^g$
  Same as above, but this time the rule imposes that no call will be marked before execution is resumed at M^g.
- $N^r \longrightarrow START_i^b \cdot M^r$
  This rule can only be applied during the marked call. By calling Π_i the run leaves the marked call, and so $START_i$ is colored black. The execution of the marked call is resumed at M, and so we have M^r.

For each execution of Δ' and each procedure call in the execution we have an execution of Δ' in which the process constants generated during the procedure call are red. The liveness problem for a local variable can now be solved applying the same technique as above; the only change is that in the definition of Δ'_r we only preserve red process constants, and in the definition of Δ'_{nd} we only remove red process constants. These are process constants that define or reference a particular incarnation of the variable. The other bitvector problems can be solved analogously.

7 Conclusion

From the perspective of program analysis, we have shown that the extension of the interprocedural setting to parallel programs does not increase the computational complexity of the operations pre* and post*. We have accommodated the extension by using *structured data* for the representation of states (here, terms over "·" and "||"; other operators than "||" might be added).

From the perspective of model checking, it comes perhaps as a surprise that the predecessor operator can be computed in linear time for a class of pushdown processes, the existing algorithms for this operator (see e.g. [3, 4, 11]) being at least cubic.

We can also look at our results as a step towards carrying automated analysis methods over from hardware to programming languages. When programs without procedures are abstracted to flow graphs, finite-state model checking methods are applicable to the transition system whose states are the nodes of the flow graph. On the other hand, in a transition system modelling procedure calls and returns, states range over an infinite domain (stacks, essentially); i.e., finite-state model checking methods are no longer applicable. (In contrast, the module notions that help to structure concurrency in hardware-like systems lead to the phenomenon of state explosion but preserve finiteness.) The present paper presents a seemingly new algorithmic principle for interprocedural analysis, in which we propose to combine a process algebraic formal framework with procedures inspired by the automata-theoretic approach to model-checking (see [10]) and by research on set-based analysis.

Future work. As pointed out in the introduction, our techniques seem to be connected to the CFL graph reachability approach of Reps [23]. This connection deserves further study. Moreover, Reps has pointed out a connection between that approach and the Dolev-Karp algorithm for *verifying cryptographic protocols* [8]. The techniques presented in this paper have potential applications also in that area.

Stacks are, of course, only one dimension of the infinity problem with program analysis/carrying model checking over from hardware to programming languages. The other dimension are data (integers, reals etc.) ranging over an infinite domain; sometimes the domain cannot be abstracted to a finite domain in an interesting way. The challenge remains to combine e.g. model checking over integers or reals as in the systems HyTech [14], Uppaal [2], DMC [7] and others with interprocedural analysis.

Acknowledgments. We thank Harald Ganzinger for discussions, and Tom Reps for encouraging us to improve the complexity bound of our algorithms.

References

[1] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science, 1990.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Petersson, W. Yi, and C. Weise. New generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer, Aaalborg, Denmark*, 1998.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR'97: Concurrency Theory*, volume 1243 of *LNCS*, pages 135–150. Springer, 1997.

[4] O. Burkart and B. Steffen. Model–checking the full modal mu–calculus for infinite sequential processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *International Colloquium on Automata, Languages, and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 419–429. Springer, 1997.

[5] W. Charatonik, D. McAllester, D. Niwinski, A. Podelski, and I. Walukiewicz. The Horn mu-calculus. In V. Pratt, editor, *Proceedings of LICS'98: Logic in Computer Science*, pages 58–69. IEEE Computer Society Press, 1998.

[6] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Proceedings of TACAS'98: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 264–289. Springer-Verlag, 1998.

[7] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Proceedings of TACAS'99: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 223–239. Springer-Verlag, 1999.

[8] D. Dolev, S. Even, and R. M. Karp. On the security of ping-pong protocols. *Information and Control*, 55(1–3):57–68, Oct./Nov./Dec. 1982.

[9] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, Oct. 1984.

[10] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural dataflow analysis. In W. Thomas, editor, *Proceedings of FoSSaCS'99: Foundations of Software Science and Computation Structures*, volume 1578 of *LNCS*, pages 14 – 30. Springer-Verlag, 1999.

[11] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. Electronic Notes in Theoretical Computer Science 9, www.elsevier.nl/locate/entcs, 13 pages, 1997.

[12] F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, Budapest, 1984.

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

[14] T. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. In *Proceedings of CAV'97*, volume 1254 of *LNCS*, pages 460–463. Springer-Verlag, 1999.

[15] J. Knoop. *Optimal Interprocedural Program Optimization: A new Framework and its Application*. PhD thesis, Univ. of Kiel, Germany, 1993. LNCS Tutorial 1428, Springer-Verlag, 1998.

[16] J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, 1996.

[17] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Bitvector analyses $\rightarrow$ No state explosion! In *Proceedings of TACAS'95*, volume 1019 of *LNCS*, pages 264 – 289. Springer-Verlag, 1995.

[18] D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. In D. Sangiorgi and R. de Simone, editors, *Proceedings of CONCUR'98: Concurrency Theory*, volume 1466 of *LNCS*, pages 50–66, 1998.

[19] D. A. McAllester. Automatic recognition of tractability in inference relations. *Journal of the ACM*, 40(2):284–303, April 1993.

[20] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*. To appear. Preliminary version in Proceedings of PEPM'97, Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, NY, 1997, pp. 74-89.

[21] M. Nivat and A. Podelski. *Tree Automata and Languages*. North-Holland, Amsterdam, 1992.

[22] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. Research Report RC 21493, IBM T.J. Watson Research Center, Yorktown Heights, NY, May 1999.

[23] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.