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 Π1 both Π1 and another procedure Π2 can be called in parallel. This spares off a new instantiation of Π2 each time Π1 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 · and ||, which are used to express stack operations and parallelism. For example, the PA-term ·((N1, N2)) — written N1 · N2 in infix notation — models the configuration with control at program node n1 that will, after the end of the current procedure, return to n2. The return itself is formally modelled by the fact that the term N1 · N2 can be rewritten (possibly in many steps) to ε · N2, where ε is a special symbol modelling termination. The PA-term (N1 || N2) · N3 models the configuration with control at program nodes n1 and n2 'in parallel' that will, after the corresponding parend, go to node n3. The term (N1 || N2) · N3 can be rewritten to (ε || ε) · N3.

As shown in [10], many bitvector problems and other distributive data flow problems can be reduced to computing the sets pre(L), 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 regular 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), Lueg and Schnoeblen prove that if a set L of PA-terms is regular then so are the sets pre(L), 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 regular language is regular if it is accepted by a tree automaton (see [12]).

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 pre(L), pre*(L), post(L), post*(L) in O(n · 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 \text{pre}^*(L) (the algorithms for the other operations are similar) 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 \text{PA} which does not have the particularly simple form of tree automata, but can be directly (and easily) derived from the definition of \text{pre}^*. In the operational step, \text{PA} 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 synchronization 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 [8]).

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 \( \text{pre}(L), \text{pre}^*(L), \text{post}(L), \text{post}^*(L) \), 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 \( \text{pre}^* \). Section 5 describes the changes needed to obtain the algorithms for the successor operator \( \text{post}^* \), the immediate-predecessor operator \( \text{pre} \) and the immediate-successor operator \( \text{post} \). Section 6 sketches the application to dataflow analysis. Section 7 presents conclusions.

## 2 Parallel Flow Graphs

The interprocedural 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 \( v := \text{exp} \) (where \( v \) is a variable, \( \text{exp} \) is an expression) or call statements of the form \( \text{call } \Pi(\text{Exp}) \) (where \( \Pi \) is a procedure identifier, and \( \text{Exp} \) is a tuple of expressions). Control flow is interpreted nondeterministically; i.e., the guards of assignments are replaced by \textbf{true}. The interprocedural 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.](image)

In a parallel FGS we also allow hyperedges of the form \( n \rightarrow \{n_1, \ldots, n_k\} \) that model a \textbf{parbegin} command, and hyperedges of the form \( \{n_1, \ldots, n_k\} \rightarrow n \), modelling a \textbf{parend} command. Wlog, we assume \( k = 2 \). We assume that the \textbf{parbegin} and \textbf{parend} hyperedges are properly nested. We do not restrict the nesting of procedure calls and \textbf{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 $\varepsilon$ using sequential composition “·” 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),

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

A PA declaration is a finite set $\Delta$ 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 $\Delta$, 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 the and the consequence below the horizontal line:

$$\begin{align*}
\Delta & \quad (X \xrightarrow{a} t) \in \Delta \\
\text{seq1} & \quad t_1 \xrightarrow{a} t_1' \quad t_1 \cdot t_2 \xrightarrow{a} t_1' \cdot t_2 \\
\text{seq2} & \quad t_2 \xrightarrow{a} t_2' \quad t_1 \cdot t_2 \xrightarrow{a} t_1 \cdot t_2' \\
\text{par1} & \quad t_1 \xrightarrow{a} t_1' \quad t_1 || t_2 \xrightarrow{a} t_1' || t_2 \\
\text{par2} & \quad t_2 \xrightarrow{a} t_2' \quad t_1 || t_2 \xrightarrow{a} t_1 || t_2'.
\end{align*}$$

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

The set $\text{pre}(L)$ of predecessors (with respect to the given PA declaration $\Delta$) 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. $t \xrightarrow{a} t'$ for some $t' \in L$. The sets $\text{pre}(L), \text{post}(L)$ and $\text{post}(L)$ of immediate predecessors, immediate successors, and successors are defined similarly.

$$\begin{align*}
\text{pre}(L) & = \{ t \mid t \xrightarrow{a} t' \text{ for some } t' \in L \} \\
\text{post}(L) & = \{ t \mid t' \xrightarrow{a} t \text{ for some } t' \in L \} \\
\text{post}(L) & = \{ t \mid t \xrightarrow{a} t' \text{ for some } t' \in L \}
\end{align*}$$

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

$$\begin{align*}
N & \rightarrow M \quad \text{for } n \rightarrow m \\
N & \rightarrow M \quad \text{for } n \xrightarrow{v} m \\
N & \rightarrow \text{START} \cdot M \quad \text{for } n \xrightarrow{\text{init}} n(T) \\
\text{END} & \rightarrow \varepsilon \quad \text{for end node of procedure } \Pi. \n\end{align*}$$

In this definition $n \xrightarrow{a} m$ means that the parallel FGS contains an edge between the program points $n$ and $m$ labeled by $l$. $N \rightarrow M$ is an abbreviation of $N \xrightarrow{a} M$ for an special “silent” action $\tau$. The definition assumes that the hyperedges $n \rightarrow \{m_1, m_2\}$ and $\{m'_1, m'_2\} \rightarrow m$ match, i.e., that they correspond to a $\text{parbegin}-\text{parend}$ instruction.

For instance, the parallel flow graph for the program

$\text{parbegin } x := 1, x := 2 \text{ parened; } y := x$

is translated into the following PA declaration.

$$\begin{align*}
\text{START} & \rightarrow K \cdot N_3 \\
K & \rightarrow N_1 || N_2 \\
N_1 & \xrightarrow{z := 1} \varepsilon \\
N_2 & \xrightarrow{z := 1} \varepsilon \\
N_3 & \xrightarrow{y := 2} \text{END} \\
\text{END} & \rightarrow \varepsilon
\end{align*}$$

A possible execution of the program is

$$\begin{align*}
\text{START} & \rightarrow K \cdot N_3 \\
& \rightarrow (N_1 || N_2) \cdot N_3 \\
& \xrightarrow{z := 2} (\varepsilon || N_2) \cdot N_3 \\
& \xrightarrow{z := 1} (\varepsilon || \varepsilon) \cdot N_3 \\
& \xrightarrow{y := 2} (\varepsilon || \varepsilon) \cdot \text{END} \\
& \rightarrow (\varepsilon || \varepsilon) \cdot \varepsilon.
\end{align*}$$

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 $\text{parbegin}-\text{parend}$ instruction the execution will be resumed at $n_3$.

We now modify the translation of the two hyperedges $n \rightarrow \{m_1, m_2\}$ and $\{m'_1, m'_2\} \rightarrow n$ as follows. We replace the rule $N \rightarrow (M_1 || M_2) \cdot M$ by two rules (using a new auxiliary symbol $K$). That is, the last case becomes:

$$\begin{align*}
N & \rightarrow K \cdot M_1, \\
K & \rightarrow M_1 || M_2 \quad \text{for } n \rightarrow \{m_1, m_2\}, \\
M'_1 & \xrightarrow{\varepsilon}, \quad \{m'_1, m'_2\} \rightarrow m. \\
M'_2 & \rightarrow \varepsilon
\end{align*}$$

Observe that now, the terms appearing in $\Delta$ are of depth 1, i.e. of the form $\varepsilon, X, X \cdot Y$ or $X || Y$. This will play a role in the complexity analysis of the algorithm to be presented next.
The problem is to compute the set \( \text{pre}^*(L) \) of predecessors of a language \( L \), wrt. a PA declaration \( \Delta \) that is derived from a parallel FGS through the translation presented in the previous section. In particular, the terms appearing in \( \Delta \) 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 \( \varepsilon \)-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 \( (\text{denoted by } P_4) \) 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 \( P_4 \) is transformed into an equivalent logic program \( \text{Red} P_4 \) 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 \( \text{Sat} P_4 \) is obtained from \( P_4 \) by means of a saturation procedure. Then, some clauses are removed from \( \text{Sat} P_4 \) to yield \( \text{Red} P_4 \). 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 \( P_4 \) 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 \( q(f) \leftarrow \text{true} \); 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 \( L_q(A) \).

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 \( q_\varepsilon \) as the initial state, the set IsNil is recognized by the tree automaton given by the three clauses below.

\[
q_\varepsilon(e)
\]

\[
q_\varepsilon(x \cdot x_2) \leftarrow q_\varepsilon(x_1), q_\varepsilon(x_2)
\]

\[
q_\varepsilon(x \parallel x_2) \leftarrow q_\varepsilon(x_1), q_\varepsilon(x_2)
\]

#### Example of a Regular Language: Atn

We define a regular set \( \text{At}_n \) of PA-terms which will be used later in Section 6. Intuitively, \( \text{At}_n \) 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
- \( t = t_1 \parallel t_2 \) and \( n \) is active at \( t_1 \) or \( t_2 \), or
- \( t = t_1 \cdot t_2 \) and \( t_1, t_2 \in \text{IsNil} \) and \( n \) is active at \( t_2 \).

We denote by \( \text{At}_n \), the set of PA-terms at which \( n \) is active. This set is a regular language; it is recognized by the tree automaton below.

\[
q(N)
\]

\[
q(x \cdot x_2) \leftarrow q(x_1)
\]

\[
q(x \parallel x_2) \leftarrow q(x_1), q_{\text{all}}(x_2)
\]

\[
q(x_1 \cdot x_2) \leftarrow q_\varepsilon(x_1), q_\varepsilon(x_2)
\]

\[
\text{all rules of the tree automaton for IsNil}
\]

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 \( q(x \cdot x_2) \leftarrow q(x_1), q_{\text{all}}(x_2) \) etc., where the predicate \( q_{\text{all}} \) stands for the state from which all terms are accepted. The definition of \( q_{\text{all}} \) requires a transition rule for each process constant.)

#### 4.2 \( \varepsilon \)-Closure

A language \( L \) of PA-terms is \( \varepsilon \)-closed if the PA-terms \( \varepsilon \cdot t, \varepsilon \parallel t \), 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 post\( ^*(L) \)) to \( \varepsilon \)-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).

\footnote{We extend this notation to general logic programs \( P \); thus, \( L_q(P) = \{ t \mid P \models q(t) \} \).}
1. The PA-terms \( t, e; t, e \) generate isomorphic transition sequences.\(^7\)

2. If the language \( L \) is \( e \)-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 \( e \)-closure.

By (1), the terms \( t, e; t, e \) and \( t, e \) are equivalent for all dataflow analysis purposes. The facts (2) and (3) guarantee that \( \text{pre}, \text{pre}^*, \text{post} \) and \( \text{post}^* \) are internal operations on regular \( e \)-closed sets.

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

\[
\begin{align*}
q(x_1 \cdot x_2) & \leftarrow q_e(x_1), q(x_2), \\
q(x_1 \parallel x_2) & \leftarrow q_e(x_1), q(x_2), \\
q(x_1 \gg x_2) & \leftarrow q_e(x_1), q(x_2).
\end{align*}
\]

(1)

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

4.3 The Declarative Part: Defining \( P_A \)

Given a PA declarative \( \Delta \) and a tree automaton \( A \) accepting an \( e \)-closed set \( L \) of PA-terms, we construct a logic program \( P_A \) with a distinguished predicate \( p_0 \) such that

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

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 \( q_0, q_1, \ldots, q_{n-1}, q_n \) (we identify \( q_0 \) 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 \( 0 \leq i, j, k \leq n \))

\[ q_e(e) \quad \text{or} \quad q_e(X) \quad \text{or} \quad q_e(x \cdot y) \leftarrow q_j(x), q_k(y) \quad \text{or} \quad q_e(x \parallel y) \leftarrow q_j(x), q_k(y). \]

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

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 \( p_0, p_1, \ldots, p_n \). Schematically:

\[ P_A = \{ \text{clauses for } q_e \text{'s in } A \} \cup \{ \text{clauses for } p_i \text{'s in Figure 2} \} \]

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 \( P_A \) if \( t \in \text{pre}^*(L) \).

---

\(^7\)They are even strongly bisimilar.

---

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

As we did with \( q_e \) and \( q_n \), we identify \( p_e \) and \( p_n \). Observe that, by assumption, the tree automaton \( A \) contains the clauses

\[ q_i(x \cdot y) \leftarrow q_i(x), q_i(y) \]

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

\[ p_i(x_1 \cdot x_2) \leftarrow p_i(x_1), p_i(x_2) \]

for every \( i = 0, \ldots, n \). Since we identify \( q_e \) 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 \( X \) as standing either for process constants or for the empty process \( e \).

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 \( X \in L_{q_i} \), then \( X \in \text{pre}^*(L_{q_i}) \);
2. If \( X \not\in \Delta \) 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 \( t_1 \in \text{pre}^*(L_{q_i}) \) and \( t_2 \in L_{q_j} \), 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 \( q_i(x_1 \cdot x_2) \leftarrow q_j(x_1), q_k(x_2) \) is a clause in \( A \) and \( t_1 \in \text{pre}^*(L_{q_i}) \) and \( t_2 \in \text{pre}^*(L_{q_j}) \) then \( t_1 \cdot t_2 \in \text{pre}^*(L_{q_i}) \).

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

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

4. If \( t_1 \in \text{pre}^*(\text{IsNil}) \) and \( t_2 \in \text{pre}^*(L_q) \) then \( t_1 \cdot t_2 \in \text{pre}^*(L_q) \).

5. If \( q_i(x_1, x_2) \leftarrow q_j(x_1), q_k(x_2) \) is a clause in \( A \) and \( t_1 \in \text{pre}^*(L_q) \) and \( t_2 \in \text{pre}^*(L_q) \), then \( t_1 \parallel t_2 \in \text{pre}^*(L_q) \).

**Proof.** We first prove that the sets \( \text{pre}^*(L_q) \) 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 \( t_1 \in \text{pre}^*(L_q) \), there is a term \( t'_1 \in L_q \) such that \( t_1 \xrightarrow{\cdot} t'_1 \). By repeated application of the rule seq1 we have \( t_1 \cdot t_2 \xrightarrow{\cdot} t'_1 \cdot t_2 \). We prove \( t'_1 \cdot t_2 \in L_q \), which implies \( t_1 \cdot t_2 \in \text{pre}^*(L_q) \). Since \( t'_1 \in L_q \) and \( t_2 \in L_q \), we have \( A \models q_j(t'_1) \) and \( A \models q_k(t_2) \). Since \( q_i(x_1, x_2) \leftarrow q_j(x_1), q_k(x_2) \) is a clause of \( A \), we also have \( A \models q_i(t'_1, t_2) \). So \( t'_1 \cdot t_2 \in L_q \).

To prove that \( \text{pre}^*(L_q) \) 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 \( \text{pre}^*(L_q) \) by \( S_i \)). We prove that for every term \( t \) and for every \( i = 0, \ldots, n \), if \( t \in \text{pre}^*(L_q) \) then \( t \in S_i \).

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

for all \( k \) for all \( t \) for all \( i \) (if \( t \xrightarrow{k} t' \in L_q \) then \( t \in S_i \)).

**Base Case:** \( k = 0 \). We proceed by structural induction on \( t \) to show:

for all \( t \) for all \( i \) (if \( t \in L_q \) then \( t \in S_i \)).

- \( t = X \in L_q \). By Condition 1, \( t \in S_i \).
- \( t = t_1 \cdot t_2 \in L_q \). There exists a clause \( q_i(x_1, x_2) \leftarrow q_j(x_1)q_k(x_2) \) in \( A \) such that \( t_1 \in L_q \) and \( t_2 \in L_q \). 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 \( j = n \).)
- \( t = t_1 \parallel t_2 \in L_q \). 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:

for all \( t \) for all \( i \) (if \( t \xrightarrow{k} t' \in L_q \) then \( t \in S_i \)).

- \( t = X \in L_q \). We assume \( X \xrightarrow{\cdot} t'' \xrightarrow{k} t' \in L_q \) for some term \( t'' \). Then, by induction hypothesis (of the induction on \( k \)), we have \( t'' \in S_i \). From \( X \xrightarrow{\cdot} t'' \) we infer \( X \xrightarrow{\cdot} t'' \in \Delta \) for some action \( a \). By Condition 2, \( X \in S_i \).
- \( t = t_1 \cdot t_2 \in L_q \). 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 \( t'_1 \in L_q \) there is a clause \( q_i(x_1, x_2) \leftarrow q_j(x_1), q_k(x_2) \) in \( A \) such that \( t'_1 \in L_q \) and \( t_2 \in L_q \). We infer \( t_1 \in S_j \) from \( t_1 \xrightarrow{k} t'_1 \in L_q \) by induction hypothesis (of the induction on \( t \)). By Condition 3, \( t_1 \parallel t_2 \in S_i \).

- \( t' = t'_1 \cdot t'_2 \) and \( t_1 \xrightarrow{k} t'_1 \) and \( t_2 \xrightarrow{k} t'_2 \) ("the rewriting in the left subterm is terminating"). Since \( t'_1 \xrightarrow{k} t'_1 \in \text{IsNil} \). Since \( t'_1, t'_2 \in L_q \), we have \( t_2 \in L_q \); 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 \( k' \) and \( k'' \), we have

\[
\begin{align*}
t'_1 &\xrightarrow{k'} t'_1, \\
t'_2 &\xrightarrow{k''} t'_2 \in \text{IsNil},
\end{align*}
\]

We apply the induction hypothesis (of the induction on \( t \)) on the fact \( t_2 \in \text{pre}^*(L_q) \) (which holds because \( t_2 \xrightarrow{k''} t'_2 \in L_q \)) and obtain \( t_2 \in S_i \). By Condition 4, \( t = t_1 \parallel t_2 \in S_i \).

\[ t = t_1 \parallel t_2 \in L_q \]. 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 \( t \in S_i \). \( \square \)

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

**Theorem 1** ("\( \text{pre}^*(q_i) = p_i \)") A PA-term \( t \) is a predecessor of some PA-term in the language \( L_0 \) 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,

\[ \text{pre}^*(L_q) = \{ t \in T_F | P_A \models p_i(t) \} \.

**Proof.** The rules of \( P_A \) model exactly the conditions defining the sets \( \text{pre}^*(L_q) \) in Proposition 2. Thus, the sets \( \{ t \in T_F | P_A \models p_i(t) \} \) are the smallest sets satisfying those conditions. Now, we only need to apply Proposition 2 in order to obtain the statement. \( \square \)

**4.4 The Operational Part:** \( P_A \rightarrow \text{Sat} P_A \rightarrow \text{Red} P_A \)

In the first stage of the operational part of the algorithm, we instantiate \( P_A \). This means that we infer all clauses of the form \( p(X) \) (where \( X \) is a process constant \( X \) of \( \Delta \) or the empty process \( \varepsilon \)) such that \( P_A \models p(X) \), and add them to \( P_A \).

The result is the saturated logic program \( \text{Sat} P_A \). Observe that the added clauses are a special case of reduction clauses.

Schematically:

\[ \text{Sat} P_A = P_A \cup \{ p(X) | P_A \models p(X) \} \]

\( ^3 \)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 \( L_q \) 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 \( \text{IsNil} \) and one deriving that the right subterm is a predecessor of a term in \( L_q \)). 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:

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

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 $pre^e(L)$, 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 $r_1$ and $r_2$ are $q_i$ or $p_i$ for some $i$ between 0 and $n$ and “$\circ$” is either “$\circ$” or “$\circ$”. If the number of clauses in $A$ is $m_A$, then the number of clauses of $P_A$ containing variables is $2m_A$ ($P_A$ contains one new clause defining $p_i$ for each ‘old’ clause defining $q_i$; see Figure 2).

We now define the logic program $P^\text{ground}_A$ as the result of replacing each clause with variables, which is necessary 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 $X \rightarrow t_1 t_2$ in $\Delta$. If $n_\Delta$ 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 $P^\text{ground}_A$ is bounded by $O(m_A \cdot n_\Delta)$.

The interest of the logic program $P^\text{ground}_A$ lies in the following proposition.

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

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

Proof. The “if” direction is trivial. For the other direction, one can show, by induction over the length of a derivation for $r(X)$ wrt. the logic program $P_A$, that each atom in the derivation is a ground instance of a clause whose head is $p(t)$; i.e., each resolution step wrt. $P_A$ is possible also wrt. $P^\text{ground}_A$. □

Proposition 3 reduces the problem of saturating $P_A$ to the problem of saturating $P^\text{ground}_A$ and then deriving all consequences of the form $r(X)$ from the set $P^\text{ground}_A$ of ground Horn clauses. This is an instance of HornSat, where the propositional constants are the atoms of the form $r(X)$ and $r(X \circ Y)$ appearing in $P^\text{ground}_A$. 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 $pre^e(L)$ of all predecessors of $L$ wrt. $\Delta$.

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 $p_i(X) \leftarrow \ldots$. This means that we have found an atom $p_i(X)$ (the one to which this clause is applied) that has a derivation wrt. $SatP_A$ (a derivation using a the clause $p_i(X) \leftarrow \ldots$ which is not a reduction clause). Thus, the atom $p_i(X)$ lies in the least model of $SatP_A$, which is equal to the least model of $P_A \models p_i(X)$. 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 $p_i(X)$. 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 $p_i(X)$. This is a contradiction. □

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

Theorem 2 Given the PA declaration $\Delta$, 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,

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

In particular, $pre^e(L)$ 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. $L = L_{q_0}(A)$ and $pre^e(L) = L_{p_0}(RedP_A)$). □

Complexity. Since $P^\text{ground}_A$ 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 $n_\Delta$ of the PA declaration $\Delta$ (i.e. the number of 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 $pre^e(L)$ is twice the number $k_A$ of states of the automaton representing the language $L$ (to compare, the tree automaton obtained in [18] has $4k_A$ states); i.e., this
bound does not depend on the input PA declaration. In contrast, the number of its clauses is bounded by \(2m_A + k_A\), where \(k_A\) is the number of process constants of \(\Delta\).

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

| \(\text{pre}^*(C)\) | (C is set of configurations, \(\text{pre}^*\) refers to flow graph system) |
| \(\text{pre}^*(L)\) | (L is \(\varepsilon\)-closure of set of PA-terms, represented by tree automaton \(A\), \(\text{pre}^*\) refers to PA declaration \(\Delta\)) |
| \(\text{lm}(P_A)\) | (\(P_A\) is a logic program (Figure 2)) |
| \(\text{lm}(Sat P_A)\) | (saturation via HornSat) |
| \(\text{lm}(Red P_A)\) | (reduction clauses of \(Sat P_A\)) |

Figure 3: Schematically, the steps of the algorithm for \(\text{pre}^*\); the PA declaration \(\Delta\) is obtained by translating a flow graph system (Section 3); the notation \(\text{lm}\) stands for ‘least model’.

5 The Algorithms for post\(^*\), pre and post

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

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

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

1. if \(X \in L_{q_i}\) then \(X \in \text{post}^*(L_{q_i})\);
2. if \(X \xrightarrow{a} t\) is a rule in \(\Delta\) and \(X \in \text{post}^*(L_{q_i})\) then \(t \in \text{post}^*(L_{q_i})\);
3. if \(q_i(x_1 \cdot x_2) \leftarrow q_i(x_1), q_i(x_2)\) is a clause in \(A\) and \(t_1 \in \text{post}^*(L_{j_1})\) and \(t_2 \in L_{k_j}\) then \(t_1 \cdot t_2 \in \text{post}^*(L_{q_i})\);
4. if \(q_i(x_1 \cdot x_2) \leftarrow q_i(x_1), q_i(x_2)\) is a clause in \(A\) and \(t_1 \in \text{post}^*(L_{j_1}) \cap \text{IsNil}\) and \(t_2 \in \text{post}^*(L_{k_j})\) then \(t_1 \cdot t_2 \in \text{post}^*(L_{q_i})\);
5. if \(q_i(x_1 \cdot x_2) \leftarrow q_i(x_1), q_i(x_2)\) is a clause in \(A\) and \(t_1 \in \text{post}^*(L_{j_1})\) and \(t_2 \in \text{post}^*(L_{k_j})\) then \(t_1 \parallel t_2 \in \text{post}^*(L_{q_i})\).

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

We define \(P_A^{\text{post}^*}\) as the logic program that consists of all reduction rules of the tree automaton \(A\) and the additional clauses in Figure 4. (Recall that \(\text{IsNil} \equiv L_{q_0}\).) Schematically:

\[
P_A^{\text{post}^*} = \{\text{clauses for } q_i\text{'s in } A\} \cup \{\text{clauses for } p_i\text{'s in Figure 4}\}
\]

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

\[
L_{p_i}(P_A^{\text{post}^*}) = \text{post}^*(L_{q_i}(A)).
\]

Figure 4: The clauses defining the predicates \(p_i\) in the logic program \(P_A^{\text{post}^*}\) for the successor operator post\(^*\); wrt. the tree automaton \(A\) with states \(q_i\) (for \(i = 0, \ldots, n\), where \(q_0 = q_a\)) and wrt. the PA declaration \(\Delta\).

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

\[
p_j(x_1), q_i(x_1)
\]

by the atom \(p_j(x_1, x_2)\) and add the following clauses defining the new predicates \(p_i^*\), for \(i = 0, \ldots, n\).

\[
p_j^*(x_1, x_2) \leftarrow q_i(x_1), p_j^*(x_2)
\]

\[
p_i^*(x_1, x_2) \leftarrow q_i(x_1), p_j^*(x_2)
\]

\[
p_i^*(x_1, x_2) \leftarrow p_j^*(x_1, x_2)
\]

\[
p_i^*(x_1, x_2) \leftarrow p_j^*(x_1, x_2)
\]

The meaning of \(p_i^*\) is given by

\[
L_{p_i^*}(P_A^{\text{post}^*}) = \text{post}^*(L_{q_i}(A)) \cap \text{IsNil}.
\]

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 \(P_A^{\text{post}^*}\) for the immediate-predecessor operator and \(P_A^{\text{post}^*}\) 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-successor 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-successor and immediate-successor relations are not transitive"); the other clauses remain the same.

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

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

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

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

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

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

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

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

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

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

6 Applications

Our algorithms can be used to solve bitvector problems and other distributive dataflow analysis problems for parallel FGSAs 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 $\Delta$. 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 $\Delta$.

$$
\text{START} \xrightarrow{\sigma_1} t_1 \xrightarrow{\sigma_2} t_2 \xrightarrow{\text{exp}(v)} t_3
$$

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 $\sigma$, 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 $START \xrightarrow{\sigma_1} t_1 \in At_n$ ("the program execution reaches a state with control at $n$")

2. No action of the form $v := \text{exp}$ is contained in the string $\sigma$ ("the variable $v$ is not defined during the execution of the statements in $\sigma$ 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 $u := \text{exp}(v)$ that stands for an assignment of an expression containing the variable $v$ ("the variable $v$ is referenced")

Let $\Delta_v$ be the subset of rewrite rules of $\Delta$ of the form

$$
X \xrightarrow{\text{exp}} t
$$

where the variable $v$ appears in the expression $\text{exp}$. Then, the set of terms $t_2$ satisfying the property described under (3) is $\text{pre}_{\Delta_v}(T_P A)$, where the immediate-predecessor operator $\text{pre}_{\Delta_v}$ refers to the reachability relation of $\Delta_v$.

Let $\Delta_{nd}$ be the subset of rewrite rules of $\Delta$ obtained by removing all rules of the form

$$
X \xrightarrow{\text{exp}} t
$$

from $\Delta$; the predecessor operator $\text{pre}_{\Delta_{nd}}^*$ refers to the reachability relation of $\Delta_{nd}$. Then, the set of terms $t_1$ such that $t_1 \xrightarrow{\sigma_1} t_2$ for some string $\sigma$ satisfying the property described under (2) is $\text{pre}_{\Delta_{nd}}(\text{pre}_{\Delta_v}(T_P A))$.

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

$$
At_n \cap \text{post}_{\Delta_v}(\{\text{START}\}) \cap \text{pre}_{\Delta_{nd}}(\text{pre}_{\Delta_v}(T_P A)).
$$

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
completes the answer).

**Local variables.** When both local and global variables are
present, it is necessary to distinguish between different in-
carnations of the same variable. This can be achieved by
translating the program into a new PA declaration $\Delta'$. The
intuition is that $\Delta'$ 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 proce-
dure; 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 live-
ness of a local variable (which is based on the existence
of an execution path) wrt. the copy. Thus, the PA declara-
tion $\Delta'$ has the same rewrite sequences of PA-terms as $\Delta$
except for the fact that, for each procedure $Pi$, at most one
call is marked. Formally, this means that a rewriting rule

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

with the special action symbol $\text{mark}$ applied at most once
for each $i$ (where $START_i$ is the process constant cor-
responding to the start node of the procedure $Pi$). Termina-
tion of the marked call is signaled by executing an action
$\text{return}$.

The restriction to at most one application to the re-
writing rule above (translating a marked procedure call) is ob-
tained by ‘coloring’ the process constants. That is, each
process constant $X$ of $\Delta$ is split in $\Delta'$ into three process con-
stants $X^g$, $X^r$, $X^b$, where the superscripts stand for green,
red, and black. The rules of $\Delta'$ 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 be-
fore the marked call starts, i.e., before the process con-
stant $\text{mark}$;

(c) black process constants can only be generated in par-
allel 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 $\text{mark}$ at most
once.

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

- $N^b \rightarrow START_i^b \cdot M^b$
  Black constants only generate black constants.

- $N^g \xrightarrow{\text{mark}} START_i \cdot M^g$
  The procedure call is marked and $START_i$ becomes
  red.

- $N^g \rightarrow START_i^r \cdot M^g$
  This rule does not mark the call of the procedure $Pi$, and by
  coloring $M$ black it guesses that some call

will be marked before execution is resumed at $M$ (no-
tice that this is only a guess, because the green con-
stant $START_i^g$ 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.

- $N^r \rightarrow START_i^r \cdot M^r$
  Same as above, but this time the rule imposes that no
call will be marked before execution is resumed at $M^r$.

- $N^r \rightarrow \text{START}_i^g \cdot M^r$
  This rule can only be applied during the marked call.
  By calling $Pi$, 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 $\Delta'$ and each procedure call in the ex-
ecution we have an execution of $\Delta'$ 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 $\Delta'$, we only preserve red process constants,
and in the definition of $\Delta_{ed}$, we only remove red process con-
stants. 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 representa-
tion of states (here, terms over ‘$\wedge$’ and ‘$\lor$’; other operators
than ‘$\equiv$’ might be added).

From the perspective of model checking, it comes per-
haps as a surprise that the predecessor operator can be com-
cuted in linear time for a class of pushdown processes, the
existing algorithms for this operator (see e.g. [5, 4, 11]) being
at least cubic.

We can also look at our results as a step towards car-
rying automated analysis methods over from hardware to
programming languages. When programs without proce-
dures are abstracted to flow graphs, finite-state model check-
ing 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, es-
tentially); 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 finite-
ness.) The present paper presents a seemingly new algorithmic
principle for interprocedural analysis, in which we pro-
pose 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 anal-
ysis.

Future work. As pointed out in the introduction, our
techniques seem to be connected to the CFL graph reacha-
bility approach of Reps [23]. This connection deserves fur-
ther 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


