Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis*

Thomas Reps[1], Stefan Schwoon[2], and Somesh Jha[1]

[1] Comp. Sci. Dept., University of Wisconsin; {reps,jha}@cs.wisc.edu
[2] Fakultät Inf., Universität Stuttgart; schwoosn@informatik.uni-stuttgart.de

Abstract. Recently, pushdown systems (PDSs) have been extended to *weighted PDSs*, in which each transition is labeled with a value, and the goal is to determine the meet-over-all-paths value (for paths that meet a certain criterion). This paper shows how weighted PDSs yield new algorithms for certain classes of interprocedural dataflow-analysis problems.

1 Introduction

This paper explores a connection between interprocedural dataflow analysis and model checking of pushdown systems (PDSs). Various connections between dataflow analysis and model checking have been established in past work, e.g., [6, 9, 23, 27, 28]; however, with one exception ([9]), past work has shed light only on the relationship between model checking and *bit-vector* dataflow-analysis problems, such as live-variable analysis and partial-redundancy elimination. In contrast, the results presented in this paper apply to (i) bit-vector problems, (ii) the one non-bit-vector problem addressed in [9], as well as (iii) certain dataflow-analysis problems that cannot be expressed as bit-vector problems, such as linear constant propagation. In general, the approach can be applied to any distributive dataflow-analysis problem for which the domain of transfer functions has no infinite descending chains. (Safe solutions are also obtained for problems that are monotonic but not distributive.)

The paper makes use of a recent result that extends PDSs to *weighted PDSs*, in which each transition is labeled with a value, and the goal is to determine the meet-over-all-paths value (for paths that meet a certain criterion) [25]. The paper shows how weighted PDSs yield new algorithms for certain classes of interprocedural dataflow-analysis problems. These ideas are illustrated by the application of weighted PDSs to linear constant propagation.

The contributions of the paper can be summarized as follows:

- Conventional dataflow-analysis algorithms merge together the values for all states associated with the same program point, regardless of the states' calling context. With the dataflow-analysis algorithm obtained via weighted PDSs, dataflow queries can be posed with respect to a regular language of stack configurations. Conventional merged dataflow information can also be obtained by issuing appropriate queries; thus, the new approach provides a strictly richer framework for interprocedural dataflow analysis than is provided by conventional interprocedural dataflow-analysis algorithms.
- Because the algorithm for solving path problems in weighted PDSs can provide a witness set of paths, it is possible to provide an explanation of why the answer to a dataflow query has the value reported.

The algorithms described in the paper have been implemented in a library that solves reachability problems on weighted PDSs [24]. The library has been used to create prototype implementations of context-sensitive interprocedural dataflow-analysis algorithms for linear constant propagation [22] and the detection of affine relationships [16]. The

* Supported by ONR contract N00014-01-1-0708.

library is available on the Internet, and may be used by third parties in the creation of dataflow-analysis tools.

The remainder of the paper is organized as follows: Section 2 introduces terminology and notation used in the paper, and defines the generalized-pushdown-reachability (GPR) framework. Section 3 presents the algorithm from [25] for solving GPR problems. Section 4 presents the new contribution of this paper—the application of the GPR framework to interprocedural dataflow analysis. Section 5 discusses related work. Appendix A describes an enhancement to the algorithm from Section 3 to generate a witness set for an answer to a GPR problem.

2 Terminology and Notation

In this section, we introduce terminology and notation used in the paper.

2.1 Pushdown Systems

A pushdown system is a transition system whose states involve a stack of unbounded length.

Definition 1. *A* **pushdown system** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P and Γ are finite sets called the* **control locations** *and the* **stack alphabet***, respectively. A* **configuration** *of $\mathcal{P}$ is a pair $\langle p, w\rangle$, where $p \in P$ and $w \in \Gamma^*$. Δ contains a finite number of* **rules** *of the form $\langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', w\rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $w \in \Gamma^*$, which define a transition relation $\Rightarrow$ between configurations of $\mathcal{P}$ as follows:*

$$\text{If } r = \langle p, \gamma\rangle \hookrightarrow_{\mathcal{P}} \langle p', w\rangle, \text{ then } \langle p, \gamma w'\rangle \xRightarrow{\langle r\rangle}_{\mathcal{P}} \langle p', ww'\rangle \text{ for all } w' \in \Gamma^*.$$

We write $c \Rightarrow_{\mathcal{P}} c'$ to express that there exists some rule r such that $c \xRightarrow{\langle r\rangle}_{\mathcal{P}} c'$; we omit the index $\mathcal{P}$ if $\mathcal{P}$ is understood. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. Given a set of configurations C, we define $pre^*(C) := \{\, c' \mid \exists c \in C \colon c' \Rightarrow^* c \,\}$ and $post^*(C) := \{\, c' \mid \exists c \in C \colon c \Rightarrow^* c' \,\}$ to be the sets of configurations that are reachable—backwards and forwards, respectively—from elements of C via the transition relation.

Without loss of generality, we assume henceforth that for every $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ we have $|w| \leq 2$; this is not restrictive because every pushdown system can be simulated by another one that obeys this restriction and is larger by only a constant factor; e.g., see [13].

Because pushdown systems have infinitely many configurations, we need some symbolic means to represent sets of configurations. We will use finite automata for this purpose.

Definition 2. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system. A $\mathcal{P}$-automaton is a quintuple $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ where $Q \supseteq P$ is a finite set of* **states***, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the set of* **transitions***, and $F \subseteq Q$ are the* **final states***. The* **initial states** *of $\mathcal{A}$ are the control locations P. A configuration $\langle p, w\rangle$ is* **accepted** *by $\mathcal{A}$ if $p \xrightarrow{w}^* q$ for some final state q. A set of configurations of $\mathcal{P}$ is* **regular** *if it is recognized by some $\mathcal{P}$-automaton. (We frequently omit the prefix $\mathcal{P}$ and simply refer to "automata" if $\mathcal{P}$ is understood.)*

A convenient property of regular sets of configurations is that they are closed under forwards and backwards reachability. In other words, given an automaton $\mathcal{A}$ that accepts the set C, one can construct automata $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ that accept $pre^*(C)$ and $post^*(C)$, respectively. The general idea behind the algorithm for pre^* [3, 8] is as follows:

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system and $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ be a $\mathcal{P}$-automaton accepting a set of configurations C. Without loss of generality we assume that $\mathcal{A}$ has no transition leading to an initial state. $pre^*(C)$ is obtained as the language of an automaton $\mathcal{A}_{pre^*} = (Q, \Gamma, \rightarrow, P, F)$ derived from $\mathcal{A}$ by a saturation procedure. The procedure adds new transitions to $\mathcal{A}$ according to the following rule:

If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and $p' \xrightarrow{w}{}^* q$ in the current automaton, add a transition (p, γ, q).

In [8] an efficient implementation of this procedure is given, which requires $\mathcal{O}(|Q|^2|\Delta|)$ time and $\mathcal{O}(|Q|\,|\Delta| + |\rightarrow_0|)$ space. Moreover, another procedure (and implementation) are presented for constructing a $\mathcal{P}$-automaton that accepts $post^*(C)$. In Section 3, we develop generalizations of these procedures. (We present these extensions for pre^*; the same basic idea applies to $post^*$, but is omitted for lack of space.)

2.2 Weighted Pushdown Systems

A weighted pushdown system is a pushdown system whose rules are given values from some domain of weights. The weight domains of interest are the bounded idempotent semirings defined in Definition 3.

Definition 3. *A* **bounded idempotent semiring** *is a quintuple $(D, \oplus, \otimes, 0, 1)$, where D is a set, 0 and 1 are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that*

1. *$(D, \oplus)$ is a commutative monoid with 0 as its neutral element, and where $\oplus$ is idempotent (i.e., for all $a \in D$, $a \oplus a = a$).*
2. *$(D, \otimes)$ is a monoid with the neutral element 1.*
3. *$\otimes$ distributes over $\oplus$, i.e. for all $a, b, c \in D$ we have*
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \qquad \textit{and} \qquad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$
4. *0 is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes 0 = 0 = 0 \otimes a$.*
5. *In the partial order $\sqsubseteq$ defined by: $\forall a, b \in D,\ a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.*

Definition 4. *A* **weighted pushdown system** *is a triple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ such that $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f\colon \Delta \rightarrow D$ is a function that assigns a value from D to each rule of $\mathcal{P}$.*

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e., if $\sigma = [r_1, \ldots, r_k]$, then we define $v(\sigma) := f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations c and c' of $\mathcal{P}$, we let $path(c, c')$ denote the set of all rule sequences $[r_1, \ldots, r_k]$ that transform c into c', i.e., $c \xRightarrow{\langle r_1 \rangle} \cdots \xRightarrow{\langle r_k \rangle} c'$

Definition 5. *Given a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$, and a regular set $C \subseteq P \times \Gamma^*$, the* **generalized pushdown reachability (GPR) problem** *is to find for each $c \in P \times \Gamma^*$:*

- *$\delta(c) := \bigoplus \{\, v(\sigma) \mid \sigma \in path(c, c'), c' \in C \,\}$;*
- *a* **witness set** *of paths $\omega(c) \subseteq \bigcup_{c' \in C} path(c, c')$ such that $\bigoplus_{\sigma \in \omega(c)} v(\sigma) = \delta(c)$.*

Notice that the extender operation $\otimes$ is used to calculate the value of a path. The value of a set of paths is computed using the combiner operation $\oplus$. In general, it is enough for $\omega(c)$ to contain only a finite set of paths whose values are minimal elements of $\{\, v(\sigma) \mid \sigma \in path(c, c'),\ c' \in C \,\}$, i.e., minimal with respect to the partial order $\sqsubseteq$ defined in Definition 3(5).

3 Solving the Generalized Pushdown Reachability Problem

This section presents the algorithm from [25] for solving GPR problems.

For the entire section, let $\mathcal{W}$ denote a fixed weighted pushdown system: $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$; let C denote a fixed regular set of configurations, represented by a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ such that $\mathcal{A}$ has no transition leading to an initial state.

The GPR problem is a multi-target meet-over-all-paths problem on a graph. The vertices of the graph are the configurations of $\mathcal{P}$, and the edges are defined by $\mathcal{P}$'s transition relation. The target vertices are the vertices in C. Both the graph and the set of target vertices can be infinite, but have some built-in structure to them; in particular, C is a regular set.

Because the GPR problem concerns infinite graphs, and not just an infinite set of paths, it differs from other work on meet-over-all-paths problems. As in the ordinary pushdown-reachability problem [3, 8], the infinite nature of the (GPR) problem is addressed by reporting the answer in an indirect fashion, namely, in the form of an (annotated) automaton. An answer automaton without its annotations is identical to an $\mathcal{A}_{pre^*}$ automaton created by the algorithm of [8]. For each $c \in pre^*(C)$, the values of $\delta(c)$ and $\omega(c)$ can be read off from the annotations by following all accepting paths for c in the automaton; for $c \notin pre^*(C)$, the values of $\delta(c)$ and $\omega(c)$ are 0 and $\emptyset$, respectively.

The algorithm is presented in several stages:

– We first define a language that characterizes the sequences of transitions that can be made by a pushdown system $\mathcal{P}$ and an automaton $\mathcal{A}$ for C.
– We then turn to weighted pushdown systems and the GPR problem. We use the language characterizations of transition sequences, together with previously known results on a certain kind of grammar problem [15, 17] to obtain a solution to the GPR problem.
– However, the solution based on grammars is somewhat inefficient; to improve the performance, we specialize the computation to our case, ending up with an algorithm for creating an annotated automaton that is quite similar to the *pre** algorithm from [8].

3.1 Languages that Characterize Transition Sequences

In this section, we make some definitions that will aid in reasoning about the set of paths that lead from a configuration c to configurations in a regular set C. We call this set the *reachability witnesses* for $c \in P \times \Gamma^*$ with respect to C: $\mathit{ReachabilityWitnesses}(c, C) = \bigcup_{c' \in C} path(c, c')$.

It is convenient to think of PDS $\mathcal{P}$ and $\mathcal{P}$-automaton $\mathcal{A}$ (for C) as being combined in sequence, to create a combined PDS, which we will call $\mathcal{PA}$. $\mathcal{PA}$'s states are $P \cup Q = Q$, and its rules are those of $\mathcal{P}$, augmented with a rule $\langle q, \gamma \rangle \hookrightarrow \langle q', \epsilon \rangle$ for each transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}$'s transition set $\rightarrow_0$.

We say that a configuration $c = \langle p, \gamma_1\gamma_2 \dots \gamma_n \rangle$ is *accepted* by $\mathcal{PA}$ if there is a path to a configuration $\langle q_f, \epsilon \rangle$ such that $q_f \in F$. Note that because $\mathcal{A}$ has no transitions leading to initial states, $\mathcal{PA}$'s behavior during an accepting run can be divided into two phases—transitions during which $\mathcal{PA}$ mimics $\mathcal{P}$, followed by transitions during which $\mathcal{PA}$ mimics $\mathcal{A}$: once $\mathcal{PA}$ reaches a state in $(Q - P)$, it can only perform a sequence of pops, possibly reaching a state in F. If the run of $\mathcal{PA}$ does reach a state in F, in terms of the features of the original $\mathcal{P}$ and $\mathcal{A}$, the second phase corresponds to automaton $\mathcal{A}$ accepting some configuration c' that has been reached by $\mathcal{P}$, starting in configuration c. In other words, $\mathcal{PA}$ accepts a configuration c iff $c \in pre^*(C)$.

The first language that we define characterizes the *pop sequences* of $\mathcal{PA}$. A pop sequence for $q \in Q$, $\gamma \in \Gamma$, and $q' \in Q$ is a sequence of $\mathcal{PA}$'s transitions that (i) starts in a configuration $\langle q, \gamma w\rangle$, (ii) ends in a configuration $\langle q', w\rangle$, and (iii) throughout the transition sequence the stack is always of the form $w'w$ for some non-empty sequence $w' \in \Gamma^+$, except in the last step, when the stack shrinks to w. Because w remains unchanged throughout a pop sequence, we need only consider pop sequences of a canonical form, i.e., those that (i) start in a configuration $\langle q, \gamma\rangle$, and (ii) end in a configuration $\langle q', \varepsilon\rangle$. The pop sequences for a given q, γ, and q' can be characterized by the complete derivation trees[3] derived from nonterminal $PS_{(q,\gamma,q')}$, using the grammar shown in Figure 1.

Production	for each
(1) $PS_{(q,\gamma,q')} \rightarrow \epsilon$	$q \xrightarrow{\gamma} q' \in \rightarrow_0$
(2) $PS_{(p,\gamma,p')} \rightarrow \epsilon$	$\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta,\ p \in P$
(3) $PS_{(p,\gamma,q)} \rightarrow PS_{(p',\gamma',q)}$	$\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta,\ p \in P,\ q \in Q$
(4) $PS_{(p,\gamma,q)} \rightarrow PS_{(p',\gamma',q')}\ PS_{(q',\gamma'',q)}$	$\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta,\ p \in P,\ q, q' \in Q$

Fig. 1. A context-free language for the pop sequences of $\mathcal{PA}$, and the $\mathcal{PA}$ rules that correspond to each production.

Theorem 1. *PDS $\mathcal{PA}$ has a pop sequence for q, γ, and q' iff nonterminal $PS_{(q,\gamma,q')}$ of the grammar shown in Figure 1 has a complete derivation tree. Moreover, for each derivation tree with root $PS_{(q,\gamma,q')}$, a preorder listing of the derivation tree's production instances (where Figure 1 defines the correspondence between productions and PDS rules) gives a sequence of rules for a pop sequence for q, γ, and q'; and every such sequence of rules has a derivation tree with root $PS_{(q,\gamma,q')}$.*

Proof (Sketch). To shrink the stack by removing the stack symbol on the left-hand side of each rule of $\mathcal{PA}$, there must be a transition sequence that removes each of the symbols that appear in the stack component of the rule's right-hand side. In other words, a pop sequence for the left-hand-side stack symbol must involve a pop sequence for each right-hand-side stack symbol.

The left-hand and right-hand sides of the productions in Figure 1 reflect the pop-sequence obligations incurred by the corresponding rule of $\mathcal{PA}$.

To capture the set *ReachabilityWitnesses*$(\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle, C)$, where C is recognized by automaton $\mathcal{A}$, we define a context-free language given by the set of productions

$$\begin{array}{lll} \textit{Accepting}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)} & \rightarrow & PS_{(p,\gamma_1,q_1)}\ PS_{(q_1,\gamma_2,q_2)}\ \ldots\ PS_{(q_{n-1},\gamma_n,q)} \\ & & \quad \text{for each } q_i \in Q, \text{ for } 1 \leq i \leq n-1; \text{ and } q \in F \\ \textit{Accepted}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)} & \rightarrow & \textit{Accepting}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)} \qquad \text{for each } q \in F \end{array}$$

This language captures all ways in which PDS $\mathcal{PA}$ can accept $\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$: the set of reachability witnesses for $\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$ corresponds to the complete derivation trees derivable from nonterminal *Accepted*$[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)}$. The subtree rooted at $PS_{(q_{i-1},\gamma_i,q_i)}$ gives the pop sequence that $\mathcal{PA}$ performs to consume symbol γ_i. (If there are no reachability witnesses for $\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$, there are no complete derivation trees with root *Accepted*$[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)}$.)

3.2 Weighted PDSs and Abstract Grammar Problems

Turning now to weighted PDSs, we will consider the weighted version of $\mathcal{PA}$, denoted by $\mathcal{WA}$, in which weighted PDS $\mathcal{W}$ is combined with $\mathcal{A}$, and each rule $\langle q, \gamma\rangle \hookrightarrow \langle q', \epsilon\rangle$

[3] A derivation tree is *complete* if it has is a terminal symbol at each leaf.

that was added due to transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}$'s transition set $\rightarrow_0$ is assigned the weight 1.

We are able to reason about semiring sums ($\oplus$) of weights on the paths that are characterized by the context-free grammars defined above using the following concept:

Definition 6. *[15, 17] Let* $(S, \sqcap)$ *be a semilattice. An* **abstract grammar** *over* $(S, \sqcap)$ *is a collection of context-free grammar productions, where each production* θ *has the form*

$$X_0 \rightarrow g_\theta(X_1, \ldots, X_k).$$

Parentheses, commas, and g_θ *(where* θ *is a production) are terminal symbols. Every production* θ *is associated with a function* $g_\theta \colon S^k \rightarrow S$*. Thus, every string* α *of terminal symbols derived in this grammar (i.e., the yield of a complete derivation tree) denotes a composition of functions, and corresponds to a unique value in* S*, which we call* $val_G(\alpha)$ *(or simply* $val(\alpha)$ *when* G *is understood). Let* $L_G(X)$ *denote the strings of terminals derivable from a nonterminal* X*. The* **abstract grammar problem** *is to compute, for each nonterminal* X*, the value*

$$m_G(X) := \bigsqcap_{\alpha \in L_G(X)} val_G(\alpha).$$

Because the complete derivation trees with root $\mathit{Accepted}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)}$ encode the transition sequences by which $\mathcal{WA}$ accepts $\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$, to cast the GPR as a grammar problem, we merely have to attach appropriate production functions to the productions so that for each rule sequence σ, and corresponding derivation tree (with yield) α, we have $v(\sigma) = val_G(\alpha)$. This is done in Figure 2: note how functions g_2, g_3, and g_4 place $f(r)$ at the beginning of the semiring-product expression; this corresponds to a preorder listing of a derivation tree's production instances (cf. Theorem 1).

Production	for each
(1) $PS_{(q,\gamma,q')} \rightarrow g_1(\epsilon)$ $g_1 = 1$	$(q, \gamma, q') \in \rightarrow_0$
(2) $PS_{(p,\gamma,p')} \rightarrow g_2(\epsilon)$ $g_2 = f(r)$	$r = \langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta,\ p \in P$
(3) $PS_{(p,\gamma,q)} \rightarrow g_3(PS_{(p',\gamma',q)})$ $g_3 = \lambda a. f(r) \otimes a$	$r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta,\ p \in P,\ q \in Q$
(4) $PS_{(p,\gamma,q)} \rightarrow g_4(PS_{(p',\gamma',q')}, PS_{(q',\gamma'',q)})$ $g_4 = \lambda a.\lambda b. f(r) \otimes a \otimes b$	$r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta,\ p \in P,\ q, q' \in Q$
(5) $\mathit{Accepting}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)} \rightarrow g_5(PS_{(p,\gamma_1,q_1)}, PS_{(q_1,\gamma_2,q_2)}, \ldots, PS_{(q_{n-1},\gamma_n,q)})$ $g_5 = \lambda a_1.\lambda a_2 \ldots \lambda a_n. a_1 \otimes a_2 \otimes \ldots \otimes a_n$	$q_i \in Q$, for $1 \leq i \leq n-1$, and $q \in F$
(6) $\mathit{Accepted}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)} \rightarrow g_6(\mathit{Accepting}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)})$ $g_6 = \lambda a.a$	$q \in F$

Fig. 2. An abstract grammar problem for the GPR problem.

To solve the GPR problem, we appeal to the following theorem:

Theorem 2. *[15, 17] The abstract grammar problem for* G *and* $(S, \sqcap)$ *can be solved by an iterative computation that finds the maximum fixed point when the following conditions hold:*

1. *The semilattice* $(S, \sqcap)$ *has no infinite descending chains.*

2. *Every production function g_θ in G is distributive, i.e.,*

$$g(\bigsqcap_{i_1 \in I_1}, \ldots, \bigsqcap_{i_k \in I_k}) = \bigsqcap_{(i_1,\ldots,i_k) \in I_1 \times \cdots \times I_k} g(x_{i_1}, \ldots, x_{i_k})$$

for arbitrary, non-empty, finite index sets $I_1, \ldots, I_k$.
3. *Every production function g_θ in G is strict in 0 in each argument.*

The abstract grammar problem given in Figure 2 meets the conditions of Theorem 2 because

1. By Definition 3, the $\oplus$ operator is associative, commutative, and idempotent; hence $(D, \oplus)$ is a semilattice. By Definition 3(5), $(D, \oplus)$ has no infinite descending chains.
2. The distributivity of each of the production functions $g_1, \ldots, g_6$ over arbitrary, non-empty, finite index sets follows from repeated application of Definition 3(3).
3. Production functions $g_3, \ldots, g_6$ are strict in 0 in each argument because 0 is an annihilator with respect to $\otimes$ (Definition 3(4)). Production functions g_1 and g_2 are constants (i.e., functions with no arguments), and hence meet the required condition trivially.

Thus, one algorithm for solving the GPR problem for a given weighted PDS $\mathcal{W}$, initial configuration $\langle p, \gamma_1\gamma_2 \ldots \gamma_n \rangle$, and regular set C (represented by automaton $\mathcal{A}$) is as follows:

- Create the combined weighted PDS $\mathcal{WA}$.
- Define the corresponding abstract grammar problem according to the schema shown in Figure 2.
- Solve this abstract grammar problem by finding the maximum fixed point using chaotic iteration: for each nonterminal X, the fixed-point-finding algorithm maintains a value $l(X)$, which is the current estimate for X's value in the maximum fixed-point solution; initially, all $l(X)$ values are set to 0; $l(X)$ is updated whenever a value $l(Y)$ changes, for any Y used on the right-hand side of a production whose left-hand-side nonterminal is X.

3.3 A More Efficient Algorithm for the GPR Problem

The approach given in the previous section is not very efficient: for a configuration $\langle p, \gamma_1\gamma_2 \ldots \gamma_n \rangle$, it takes $\Theta(|Q|^{n-1}|F|)$ time and space just to create the grammar productions in Figure 2 with left-hand-side nonterminal *Accepting*$[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)}$. However, we can improve on the algorithm of the previous section because not all instantiations of the productions listed in Figure 2 are relevant to the final solution; we want to prevent the algorithm from exploring useless nonterminals of the grammar shown in Figure 2.

Moreover, all GPR questions with respect to a given target-configuration set C involve the same subgrammar for the *PS* nonterminals. As in the (ordinary) pushdown-reachability problem [3, 8], the information about whether a complete derivation tree with root nonterminal $PS_{(q,\gamma,q')}$ exists (i.e., whether $PS_{(q,\gamma,q')}$ is a *productive* nonterminal) can be precomputed and returned in the form of an (annotated) automaton of size $\mathcal{O}(|Q|\,|\Delta| + |\rightarrow_0|)$. Exploring the *PS* subgrammar lazily saves us from having to construct the entire *PS* subgrammar. Productive nonterminals represent automaton transitions, and the productions that involve any given transition can be constructed on-the-fly, as is done in Algorithm 1, shown in Figure 3.

It is relatively straightforward to see that Algorithm 1 solves the grammar problem for the *PS* subgrammar from Figure 2: *workset* contains the set of transitions

Algorithm 1

Input: a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$,
where $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$;
a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \to_0, P, F)$ that accepts C,
such that $\mathcal{A}$ has no transitions into P states.

Output: a $\mathcal{P}$-automaton $\mathcal{A}_{pre^*} = (Q, \Gamma, \to, P, F)$ that accepts $pre^*(C)$;
a function l that maps every $(q, \gamma, q') \in \to$ to the value of $m_G(PS_{(q,\gamma,q')})$
in the abstract grammar problem defined in Figure 2.

1 **procedure** *update*(t, r, T)
2 **begin**
3 $\quad \to := \to \cup \{t\}$;
4 $\quad l(t) := l(t) \oplus (f(r) \otimes l(T(1)) \otimes \ldots \otimes l(T(|T|)))$;
5 $\quad$ **if** $l(t)$ changed value **then** $workset := workset \cup \{t\}$
6 **end**
7
8 $\to := \to_0$; $l \equiv 0$; $workset := \to_0$;
9 **for all** $t \in \to_0$ **do** $l(t) := 1$;
10 **for all** $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $update((p, \gamma, p'), r, ())$;
11 **while** $workset \neq \emptyset$ **do**
12 $\quad$ select and remove a transition $t = (q, \gamma, q')$ from $workset$;
13 $\quad$ **for all** $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in \Delta$ **do** $update((p_1, \gamma_1, q'), r, (t))$;
14 $\quad$ **for all** $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta$ **do**
15 $\qquad$ **for all** $t' = (q', \gamma_2, q'') \in \to$ **do** $update((p_1, \gamma_1, q''), r, (t, t'))$;
16 $\quad$ **for all** $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \gamma_2\gamma \rangle \in \Delta$ **do**
17 $\qquad$ **if** $t' = (p', \gamma_2, q) \in \to$ **then** $update((p_1, \gamma_1, q'), r, (t', t))$;
18 **return** $((Q, \Gamma, \to, P, F), l)$

Fig. 3. An on-the-fly algorithm for solving the grammar problem for the *PS* subgrammar from Figure 2.

(*PS* nonterminals) whose value $l(t)$ has been updated since it was last considered; in line 8 all values are set to 0. A function call *update*(t, r, T) computes the new value for transition t if t can be created using rule r and the transitions in the ordered list T. Lines 9–10 process the rules of types (1) and (2), respectively. Lines 11–17 represent the fixed-point-finding loop: line 13, 15, and 17 simulate the processing of rules of types (3) and (4) that involve transition t on their right-hand side; in particular, line 4 corresponds to invocations of production functions g_3 and g_4. Note that line 4 can change $l(t)$ only to a smaller value (w.r.t. $\sqsubseteq$). The iterations continue until the values of all transitions stabilize, i.e., $workset$ is empty.

From the fact that Algorithm 1 is simply a different way of expressing the grammar problem for the *PS* subgrammar, we know that the algorithm terminates and computes the desired result. Moreover, apart from operations having to do with l, the algorithm is remarkably similar to the pre^* algorithm from [8]—the only major difference being that transitions are stored in a workset and processed multiple times, whereas in [8] each transition is processed exactly once. Thus, the time complexity increases from the $\mathcal{O}(|Q|^2|\Delta|)$ complexity of the unweighted case [8] by a factor that is no more than the length of the maximal-length descending chain in the semiring.

Given the annotated pre^* automaton, the value of $\delta(c)$ for any configuration c can be read off from the automaton by following all paths by which c is accepted—accumulating a value for each path—and taking the meet of the resulting value set. The value-accumulation step can be performed using a straightforward extension of a standard algorithm for simulating an NFA (cf. [1, Algorithm 3.4]).

Algorithm 1 is a dynamic-programming algorithm for determining $\delta(c)$; Appendix A describes how to extend Algorithm 1 to keep additional annotations on transitions so that the path set $\omega(c)$ can be obtained.

4 Applications to Interprocedural Dataflow Analysis

This section describes the application of weighted PDSs to interprocedural dataflow analysis, and shows that the algorithm from Section 3 provides a way to generalize previously known frameworks for interprocedural dataflow analysis [22, 26]. The running example used in this section illustrates the application of the approach to linear constant propagation [22]. G. Balakrishnan has also used the approach to implement an interprocedural dataflow-analysis algorithm due to M. Müller-Olm and H. Seidl, which determines, for each program point n, the set of all affine relations that hold among program variables whenever n is executed [16].

Interprocedural Dataflow Analysis, Supergraphs, and Exploded Supergraphs

Interprocedural dataflow-analysis problems are often defined in terms of a program's *supergraph*, an example of which is shown in Figure 4. A supergraph consists of a

```
int x;

void main() {
     n1: x = 5;
  n2,n3: p();
         return;
}

void p() {
   n4: if (...) {
         n5: x = x + 1;
      n6,n7: p();
         n8: x = x - 1;
  }
  n9: else if (...) {
         n10: x = x - 1;
    n11,n12: p();
         n13: x = x + 1;
  }
  return;
}
```

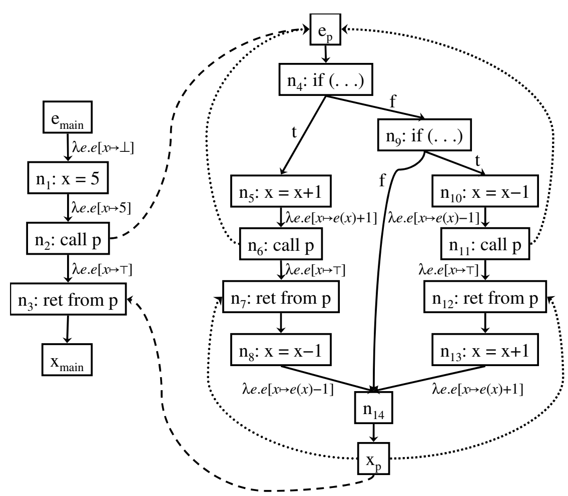

Fig. 4. A program fragment and its supergraph. The environment transformer for all unlabeled edges is $\lambda e.e$.

collection of control-flow graphs—one for each procedure—one of which represents the program's main procedure. The flowgraph for a procedure *p* has a unique *enter* node, denoted by e_p, and a unique *exit* node, denoted by x_p. The other nodes of the flowgraph represent statements and conditions of the program in the usual way,[4] except that each procedure call in the program is represented in the supergraph by two nodes, a *call* node and a *return-site* node (e.g., see the node-pairs (n_2, n_3), (n_6, n_7), (n_{11}, n_{12}) in Figure 4). In addition to the ordinary intraprocedural edges that connect the nodes of the individual control-flow graphs, for each procedure call—represented, say, by call node c and return-site node r—the supergraph contains three edges: an intraprocedural *call-to-return-site* edge from c to r; an interprocedural *call-to-enter* edge from c to the enter node of the called procedure; an interprocedural *exit-to-return-site* edge from the exit node of the called procedure to r.

Definition 7. *A* **path** *of length j from node m to node n is a (possibly empty) sequence of j edges, which will be denoted by $[e_1, e_2, \ldots, e_j]$, such that the source of e_1 is m, the target of e_j is n, and for all i, $1 \leq i \leq j-1$, the target of edge e_i is the source of edge e_{i+1}. Path concatenation is denoted by $\|$.*

The notion of an *(interprocedurally) valid path* is necessary to capture the idea that not all paths in a supergraph represent potential execution paths. A valid path is one that respects the fact that a procedure always returns to the site of the most recent call. We distinguish further between a *same-level valid path*—a path that starts and ends in the same procedure, and in which every call has a corresponding return (and vice versa)—and a *valid path*—a path that may include one or more unmatched calls:

Definition 8. *The sets of* **same-level valid paths** *and* **valid paths** *in a supergraph are defined inductively as follows:*

– *The empty path is a* **same-level valid path** *(and therefore a* **valid path***).*
– *Path $p \;\|\; [e]$ is a* **valid path** *if either (i) e is not an exit-to-return-site edge and p is a valid path, or (ii) e is an exit-to-return-site edge and $p = p_h \;\|\; [e_c] \;\|\; p_t$, where p_t is a same-level valid path, p_h is a valid path, and the source node of e_c is the call node that matches the return-site node at the target of e. Such a path is a* **same-level valid path** *if p_h is also a same-level valid path.*

Example 1. In the supergraph shown in Figure 4, the path

$$e_{main} \to n_1 \to n_2 \to e_p \to n_4 \to n_9 \to n_{14} \to x_p \to n_3$$

is a (same-level) valid path; the path

$$e_{main} \to n_1 \to n_2 \to e_p \to n_4 \to n_9$$

is a (non-same-level) valid path because the call-to-start edge $n_2 \to e_p$ has no matching exit-to-return-site edge; the path

$$e_{main} \to n_1 \to n_2 \to e_p \to n_4 \to n_9 \to n_{14} \to x_p \to n_7$$

is not a valid path because the exit-to-return-site edge $x_p \to n_7$ does not correspond to the preceding call-to-start edge $n_2 \to e_p$.

A context-sensitive interprocedural dataflow analysis is one in which the analysis of a called procedure is "sensitive" to the context in which it is called. A context-sensitive

[4] The nodes of a flowgraph can represent individual statements and conditions; alternatively, they can represent basic blocks.

analysis captures the fact that calls on a procedure that arrive via different calling contexts can cause different sets of execution states to arise on entry to a procedure. More precisely, the goal of a context-sensitive analysis is to find the *meet-over-all-valid-paths* value for nodes of a supergraph [14, 22, 26].

The remainder of this section considers the Interprocedural Distributive Environment (IDE) framework for context-sensitive interprocedural dataflow analysis [22]. It applies to problems in which the dataflow information at a program point is represented by a finite environment (*i.e.*, a mapping from a finite set of *symbols* to a finite-height domain of *values*), and the effect of a program operation is captured by an "environment-transformer function" associated with each supergraph edge. The transformer functions are assumed to distribute over the meet operation on environments.

Two IDE problems are (decidable) variants of the constant-propagation problem: *copy-constant propagation* and *linear-constant propagation*. The former interprets assignment statements of the form $x = 7$ and $y = x$. The latter also interprets statements of the form $y = -2 * x + 5$.

By means of an "explosion transformation", an IDE problem can be transformed from a path problem on a program's supergraph to a path problem on a graph that is *larger*, but in which every edge is labeled with a much *simpler* edge function (a so-called "micro-function") [22]. Each micro-function on an edge $d_1 \rightarrow d_2$ captures the effect that the value of symbol d_1 in the argument environment has on the value of symbol d_2 in the result environment. Figure 5 shows the exploded representations of four environment-transformer functions used in linear constant propagation. Figure 5(a)

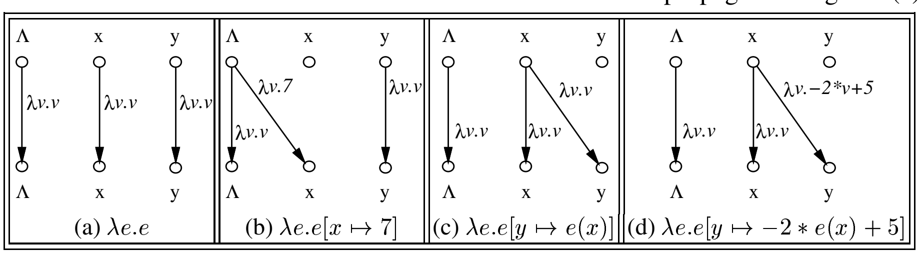

Fig. 5. The exploded representations of four environment-transformer functions used in linear constant propagation.

shows how the identity function $\lambda e.e$ is represented. Figure 5(b)–Figure 5(d) show the representations of the functions $\lambda e.e[x \mapsto 7]$, $\lambda e.e[y \mapsto e(x)]$, and $\lambda e.e[y \mapsto -2 * e(x) + 5]$, which are the dataflow functions for the assignment statements $x = 7, y = x$, and $y = -2 * x + 5$, respectively. (The Λ vertices are used to represent the effects of a function that are independent of the argument environment. Each graph includes an edge of the form $\Lambda \rightarrow \Lambda$, labeled with $\lambda v.v$; these edges are needed to capture function composition properly [22].)

Figure 6 shows the exploded supergraph that corresponds to the program from Figure 4 for the linear constant-propagation problem.

From Exploded Supergraphs to Weighted PDSs

We now show how to solve linear constant-propagation problems in a context-sensitive fashion by defining a generalized pushdown reachability problem in which the paths of the (infinite-state) transition system correspond to valid paths in the exploded supergraph from (e_{main}, Λ). To do this, we encode the exploded supergraph as a weighted PDS whose weights are drawn from a semiring whose value set is the set of functions

$$F_{lc} = \{\lambda l.\top\} \cup \{\lambda l.(a * l + b) \sqcap c \mid a \in \mathbb{Z}, b \in \mathbb{Z}, \text{and } c \in \mathbb{Z}_{\perp}^{\top}\}.$$

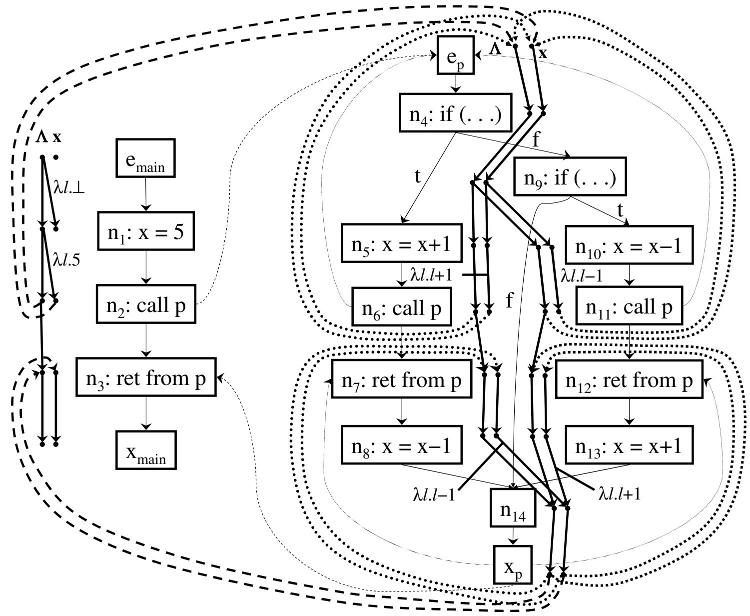

Fig. 6. The exploded supergraph of the program from Figure 4 for the linear constant-propagation problem. The micro-functions are all *id*, except where indicated.

Every function $f \in F_{lc} - \{\lambda l.\top\}$ can be represented by a triple (a, b, c), where $a \in \mathbb{Z}$, $b \in \mathbb{Z}, c \in \mathbb{Z}_{\perp}^{\top}$, and

$$f = \lambda l. \begin{cases} \top & \text{if } l = \top \\ (a * l + b) \sqcap c & \text{otherwise} \end{cases}$$

The third component c is needed so that the meet of two functions can be represented. (See [22] for details.) The semiring value 0 is $\lambda l.\top$; the semiring value 1 is the identity function, whose representation is $(1, 0, \top)$. We also denote the identity function by *id*. By convention, a constant function $\lambda l.b$ is represented as $(0, b, \top)$.

The operations $\oplus$ and $\otimes$ are defined as follows:

$$(a_2, b_2, c_2) \oplus (a_1, b_1, c_1) = \begin{cases} (a_1, b_1, c_1 \sqcap c_2) & \text{if } a_1 = a_2 \text{ and } b_1 = b_2 \\ (a_1, b_1, c) & \text{where } c = (a_1 * l_0 + b_1) \sqcap c_1 \sqcap c_2, \\ & \text{if } l_0 = (b_1 - b_2)/(a_2 - a_1) \in \mathbb{Z} \\ (1, 0, \perp) & \text{otherwise} \end{cases}$$

$$(a_2, b_2, c_2) \otimes (a_1, b_1, c_1) = ((a_1 * a_2), (a_1 * b_2 + b_1), ((a_1 * c_2 + b_1) \sqcap c_1)).$$

Here it is assumed that $x * \top = \top * x = x + \top = \top + x = \top$ for $x \in \mathbb{Z}_{\perp}^{\top}$ and that $x * \perp = \perp * x = x + \perp = \perp + x = \perp$ for $x \in \mathbb{Z}_{\perp}$. The second case for the combiner operator is obtained by equating the terms $a_1 * y + b_1$ and $a_2 * y + b_2$ and taking the solution for y, provided it is integral.

The control locations correspond to the program's variables (and also to Λ). Stack symbols, such as n_4, n_5 and n_6, correspond to nodes of the supergraph.

With one exception, each edge in the exploded supergraph corresponds to one rule of the weighted PDS. The encoding can be described in terms of the kinds of edges that occur in the supergraph.

A few of the weighted PDS's rules for the (exploded) intraprocedural edges are as follows:

Intraprocedural edges in main		Intraprocedural edges in p	
$\langle \Lambda, n_1 \rangle \hookrightarrow \langle \Lambda, n_2 \rangle$	*id*	$\langle \Lambda, n_4 \rangle \hookrightarrow \langle \Lambda, n_5 \rangle$	*id*
$\langle \Lambda, n_1 \rangle \hookrightarrow \langle x, n_2 \rangle$	$\lambda l.5$	$\langle x, n_4 \rangle \hookrightarrow \langle x, n_5 \rangle$	*id*
$\langle \Lambda, n_3 \rangle \hookrightarrow \langle \Lambda, x_{main} \rangle$	*id*	$\langle \Lambda, n_5 \rangle \hookrightarrow \langle \Lambda, n_6 \rangle$	*id*
$\langle x, n_3 \rangle \hookrightarrow \langle x, x_{main} \rangle$	*id*	$\langle x, n_5 \rangle \hookrightarrow \langle x, n_6 \rangle$	$\lambda l.l+1$

In a rule such as

$$\langle x, n_5 \rangle \hookrightarrow \langle x, n_6 \rangle \qquad \lambda l.l+1 \tag{1}$$

the second component of each tuple implies that the currently active procedure is *p*, and the rest of the stack is not changed.

At each call site, each PDS rule that encodes an edge in the exploded representation of a call-to-enter edge has two stack symbols on its right-hand side. The second symbol is the name of the corresponding return-site node, which is pushed on the stack:

Transitions for call site n_2		Transitions for call site n_6		Transitions for call site n_{11}	
$\langle \Lambda, n_2 \rangle \hookrightarrow \langle \Lambda, e_p\ n_3 \rangle$	*id*	$\langle \Lambda, n_6 \rangle \hookrightarrow \langle \Lambda, e_p\ n_7 \rangle$	*id*	$\langle \Lambda, n_{11} \rangle \hookrightarrow \langle \Lambda, e_p\ n_{12} \rangle$	*id*
$\langle x, n_2 \rangle \hookrightarrow \langle x, e_p\ n_3 \rangle$	*id*	$\langle x, n_6 \rangle \hookrightarrow \langle x, e_p\ n_7 \rangle$	*id*	$\langle x, n_{11} \rangle \hookrightarrow \langle x, e_p\ n_{12} \rangle$	*id*

The process of returning from *p* is encoded by popping the topmost stack symbols off the stack.

Transitions to return from p	
$\langle \Lambda, x_p \rangle \hookrightarrow \langle \Lambda, \varepsilon \rangle$	*id*
$\langle x, x_p \rangle \hookrightarrow \langle x, \varepsilon \rangle$	*id*

Obtaining Dataflow Information from the Exploded Supergraph's Weighted PDS

For linear constant propagation, we are interested in a generalized reachability problem from configuration $\langle \Lambda, e_{main} \rangle$. Thus, to obtain dataflow information from the exploded supergraph's weighted PDS, we perform the following steps:

– Define a regular language R for the configurations of interest. This can be done by creating an automaton for R, and giving each edge of the automaton the weight *id*.
– Apply Algorithm 1 to create a weighted automaton for $pre^*(R)$.
– Inspect the $pre^*(R)$-automaton to find the transition $\Lambda \xrightarrow{e_{main}} accepting\_state$. Return the weight on this transition as the answer.

In the following, we often write $\langle x, \alpha \rangle$, where α is a regular expression, to mean the set of all configurations $\langle x, w \rangle$ where w is in the language of stack contents defined by α.

Example 2. For the query $pre^*(\langle x, e_p\ (n_{12}\ n_7)^*\ n_3 \rangle)$, the semiring value associated with the configuration $\langle \Lambda, e_{main} \rangle$ is $\lambda l.5$, which means that the value of program variable x must be 5 whenever *p* is entered with a stack of the form "$e_p\ (n_{12}\ n_7)^*\ n_3$"; i.e., *main* called *p*, which then called itself recursively an arbitrary number of times, alternating between the two recursive call sites.

A witness-path set for the configuration $\langle \Lambda, e_{main} \rangle$ is a singleton set, consisting of the following path:

Semiring value	Configuration	Rule	Rule weight
$\lambda l.5$	$\langle \Lambda, e_{main} \rangle$	$\langle \Lambda, e_{main} \rangle \hookrightarrow \langle \Lambda, n_1 \rangle$	*id*
$\lambda l.5$	$\langle \Lambda, n_1 \rangle$	$\langle \Lambda, n_1 \rangle \hookrightarrow \langle x, n_2 \rangle$	$\lambda l.5$
id	$\langle x, n_2 \rangle$	$\langle x, n_2 \rangle \hookrightarrow \langle x, e_p\ n_3 \rangle$	*id*
id	$\langle x, e_p\ n_3 \rangle$	Configuration accepted by query automaton	

Example 3. One example of a situation in which the stack is of the form $e_p\ (n_{12}\ n_7)^*\ n_3$ is when *main* calls p at n_2 (n_3); p calls p at n_6 (n_7); and finally p calls p at n_{11} (n_{12}). In this case, the stack contains $e_p\ n_{12}\ n_7\ n_3$. As expected, for the query $pre^*(\langle x, e_p\ n_{12}\ n_7\ n_3\rangle)$, the semiring value associated with the configuration $\langle \Lambda, e_{main}\rangle$ is $\lambda l.5$.

In this case, a witness-path set for the configuration $\langle \Lambda, e_{main}\rangle$ is a singleton set, consisting of the following path:

Semiring value	Configuration	Rule	Rule weight
$\lambda l.5$	$\langle \Lambda, e_{main}\rangle$	$\langle \Lambda, e_{main}\rangle \hookrightarrow \langle \Lambda, n_1\rangle$	*id*
$\lambda l.5$	$\langle \Lambda, n_1\rangle$	$\langle \Lambda, n_1\rangle \hookrightarrow \langle x, n_2\rangle$	$\lambda l.5$
id	$\langle x, n_2\rangle$	$\langle x, n_2\rangle \hookrightarrow \langle x, e_p\ n_3\rangle$	*id*
id	$\langle x, e_p\ n_3\rangle$	$\langle x, e_p\rangle \hookrightarrow \langle x, n_4\rangle$	*id*
id	$\langle x, n_4\ n_3\rangle$	$\langle x, n_4\rangle \hookrightarrow \langle x, n_5\rangle$	*id*
id	$\langle x, n_5\ n_3\rangle$	$\langle x, n_5\rangle \hookrightarrow \langle x, n_6\rangle$	$\lambda l.l+1$
$\lambda l.l-1$	$\langle x, n_6\ n_3\rangle$	$\langle x, n_6\rangle \hookrightarrow \langle x, e_p\ n_7\rangle$	*id*
$\lambda l.l-1$	$\langle x, e_p\ n_7\ n_3\rangle$	$\langle x, e_p\rangle \hookrightarrow \langle x, n_4\rangle$	*id*
$\lambda l.l-1$	$\langle x, n_4\ n_7\ n_3\rangle$	$\langle x, n_4\rangle \hookrightarrow \langle x, n_9\rangle$	*id*
$\lambda l.l-1$	$\langle x, n_9\ n_7\ n_3\rangle$	$\langle x, n_9\rangle \hookrightarrow \langle x, n_{10}\rangle$	*id*
$\lambda l.l-1$	$\langle x, n_{10}\ n_7\ n_3\rangle$	$\langle x, n_{10}\rangle \hookrightarrow \langle x, n_{11}\rangle$	$\lambda l.l-1$
id	$\langle x, n_{11}\ n_7\ n_3\rangle$	$\langle x, n_{11}\rangle \hookrightarrow \langle x, e_p\ n_{12}\rangle$	*id*
id	$\langle x, e_p\ n_{12}\ n_7\ n_3\rangle$	Configuration accepted by query automaton	

Notice that the witness-path set for the configuration $\langle \Lambda, e_{main}\rangle$ is more complicated in the case of the query $pre^*(\langle x, e_p\ n_{12}\ n_7\ n_3\rangle)$ than in the case of the query $pre^*(\langle x, e_p\ (n_{12}\ n_7)^*\ n_3\rangle)$, even though the latter involves a regular operator.

Example 4. Conventional dataflow-analysis algorithms merge together (via meet, i.e., $\oplus$) the values for each program point, regardless of calling context. The machinery described in this paper provides a strict generalization of conventional dataflow analysis because the merged information can be obtained by issuing an appropriate query.

For instance, the value that the algorithms given in [14, 22, 26] would obtain for the tuple $\langle x, e_p\rangle$ can be obtained via the query $pre^*(\langle x, e_p\ (n_7 + n_{12})^*\ n_3\rangle)$. When we perform this query, the semiring value associated with the configuration $\langle \Lambda, e_{main}\rangle$ is $\lambda l.\bot$. This means that the value of program variable x may not always be the same when p is entered with a stack of the form "$e_p\ (n_7 + n_{12})^*\ n_3$". For this situation, a witness-path set for the configuration $\langle \Lambda, e_{main}\rangle$ consists of two paths, which share the first four configurations; the semiring value associated with $\langle x, e_p\ n_3\rangle$ is $\lambda l.\bot = id \oplus \lambda l.l-1$:

Semiring value	Configuration	Rule	Rule weight
$\lambda l.\bot$	$\langle \Lambda, e_{main}\rangle$	$\langle \Lambda, e_{main}\rangle \hookrightarrow \langle \Lambda, n_1\rangle$	*id*
$\lambda l.\bot$	$\langle \Lambda, n_1\rangle$	$\langle \Lambda, n_1\rangle \hookrightarrow \langle x, n_2\rangle$	$\lambda l.5$
$\lambda l.\bot$	$\langle x, n_2\rangle$	$\langle x, n_2\rangle \hookrightarrow \langle x, e_p\ n_3\rangle$	*id*
$\lambda l.\bot$	$\langle x, e_p\ n_3\rangle$		
id	$\langle x, e_p\ n_3\rangle$	Configuration accepted by query automaton	
$\lambda l.l-1$	$\langle x, e_p\ n_3\rangle$	$\langle x, e_p\rangle \hookrightarrow \langle x, n_4\rangle$	*id*
$\lambda l.l-1$	$\langle x, n_4\ n_3\rangle$	$\langle x, n_4\rangle \hookrightarrow \langle x, n_9\rangle$	*id*
$\lambda l.l-1$	$\langle x, n_9\ n_3\rangle$	$\langle x, n_9\rangle \hookrightarrow \langle x, n_{10}\rangle$	*id*
$\lambda l.l-1$	$\langle x, n_{10}\ n_3\rangle$	$\langle x, n_{10}\rangle \hookrightarrow \langle x, n_{11}\rangle$	$\lambda l.l-1$
id	$\langle x, n_{11}\ n_3\rangle$	$\langle x, n_{11}\rangle \hookrightarrow \langle x, e_p\ n_{12}\rangle$	*id*
id	$\langle x, e_p\ n_{12}\ n_3\rangle$	Configuration accepted by query automaton	

The Complexity of the Dataflow-Analysis Algorithm

Let E denote the number of edges in the supergraph, and let Var denote the number of symbols in the domain of an environment. The encoding of an exploded supergraph as a PDS leads to a PDS with Var control locations and $|\Delta| = E \cdot Var$ rules. If R is the regular language of configurations of interest, assume that R can be encoded by a weighted automaton with $|Q| = s + Var$ states and t transitions. Let l denote the maximal length of a descending chain in the semiring formed by the micro-functions.

The cost of a pre^* query to obtain dataflow information for R is therefore no worse than $\mathcal{O}(s^2 \cdot Var \cdot E \cdot l + Var^3 \cdot E \cdot l)$ time and $\mathcal{O}(s \cdot Var \cdot E + Var^2 \cdot E + t)$ space, according to the results of Section 3 and [8].

How Clients of Dataflow Analysis Can Take Advantage of this Machinery

Algorithm 1 and the construction given above provide a new algorithm for interprocedural dataflow analysis. As demonstrated by Examples 2, 3, and 4, with the weighted-PDS machinery, dataflow queries can be posed with respect to a regular language of initial stack configurations, which provides a strict generalization of the kind of queries that can be posed using ordinary interprocedural dataflow-analysis algorithms.

For clients of interprocedural dataflow analysis, such as program optimizers and tools for program understanding, this offers the ability to provide features that were previously unavailable:

– A program optimizer could make a query about dataflow values according to a possible pattern of inline expansions. This would allow the optimizer to determine—*without first performing an explicit expansion*—whether the inline expansion would produce favorable dataflow values that would allow the code to be optimized.
– A tool for program understanding could let users pose queries about dataflow information with respect to a regular language of initial stack configurations.

The first of these possibilities is illustrated by Figure 7, which shows a transformed version of the program from Figure 4. The transformed program takes advantage of the information obtained from Example 2, namely, that in Figure 4 the value of x is 5 whenever p is entered with a stack of the form "$e_p\ (n_{12}\ n_7)^*\ n_3$". In the transformed program, all calls to p that mimic the calling pattern "$(n_{12}\ n_7)^*\ n_3$" (from the original program) are replaced by calls to p'. In p', a copy of p has been inlined (and simplified) at the first recursive call site. Whenever the calling pattern fails to mimic "$(n_{12}\ n_7)^*\ n_3$", the original procedure p is called instead.

5 Related Work

Several connections between dataflow analysis and model checking have been established in past work [27, 28, 23, 6]. The present paper continues this line of inquiry, but makes two contributions:

– Previous work addressed the relationship between model checking and *bit-vector* dataflow-analysis problems, such as live-variable analysis and partial-redundancy elimination. In this paper, we show how a technique inspired by one developed in the model-checking community [3, 8]—but generalized from its original form [25]—can be applied to certain dataflow-analysis problems that cannot be expressed as bit-vector problems.
– Previous work has used temporal-logic expressions to specify dataflow-analysis problems. This paper's results are based on a more basic model-checking primitive, namely pre^*. (The approach also extends to $post^*$.)

These ideas have been illustrated by applying them to linear constant propagation, which is not expressible as a bit-vector problem.

```
int x;                          void p'() {
                                  if (...) {
void main() {                        if (...) {  // Inlined call n6,n7
  x = 5;                                 x = 7;
  p'();                                  p();  // n6,n7; n6,n7
  return;                            }
}                                    else if (...) {
                                         p'(); // n6,n7; n11,n12
void p() {                           } // End inlined call n6,n7
  if (...) {                      }
     x = x + 1;                   else if (...) {
     p();                            x = 4;
     x = x - 1;                      p();
  }                               }
  else if (...) {                 x = 5;
     x = x - 1;                   return;
     p();                       }
     x = x + 1;
  }
  return;
}
```

Fig. 7. A transformed version of the program from Figure 4 that takes advantage of the fact that in Figure 4 the value of x is 5 whenever p is entered with a stack of the form "$e_p\ (n_{12}\, n_7)^*\ n_3$".

Bouajjani, Esparza, and Toulli [4] independently developed a similar framework, in which pre^* and $post^*$ queries on pushdown systems with weights drawn from a semiring are used to solve (overapproximations of) reachability questions on concurrent communicating pushdown systems. Their method of obtaining weights on automaton transitions significantly differs from ours. Instead of deriving the weights directly, they are obtained using a fixpoint computation on a matrix whose entries are the transitions of the pre^* automaton. This allows them to obtain weights even when the semiring does have infinite descending chains (provided the extender operator is commutative), but leads to a less efficient solution for the finite-chain case. In the latter case, in the terms of Section 4, their algorithm has time complexity $\mathcal{O}(((s+Var)\cdot E\cdot Var+t)^2\cdot E\cdot Var\cdot(s+Var)\cdot l)$, i.e., proportional to Var^6 and E^3. All but one of the semirings used in [4] have only finite descending chains, so Algorithm 1 applies to those cases and provides a more efficient solution.

The most closely related papers in the dataflow-analysis literature are those that address demand-driven interprocedural dataflow analysis.

– Reps [19, 18] presented a way in which algorithms that solve demand versions of interprocedural analysis problems can be obtained automatically from their exhaustive counterparts (expressed as logic programs) by making use of the "magic-sets transformation" [2], which is a general transformation developed in the logic-programming and deductive-database communities for creating efficient demand versions of (bottom-up) logic programs, and/or tabulation [29], which is another method for efficiently evaluating recursive queries in deductive databases. This approach was used to obtain demand algorithms for interprocedural bit-vector problems.

– Subsequent work by Reps, Horwitz, and Sagiv extended the logic-programming approach to the class of IFDS problems [20].[5] They also gave an explicit demand algorithm for IFDS problems that does not rely on the magic-sets transformation [11].
– Both exhaustive and demand algorithms for solving a certain class of IDE problems are presented in [22]. The relationship between the two algorithms given in that paper is similar to the relationship between the exhaustive [20] and demand [11] algorithms for IFDS problems.
– A fourth approach to obtaining demand versions of interprocedural dataflow-analysis algorithms was investigated by Duesterwald, Gupta, and Soffa [7]. In their approach, for each query a collection of dataflow equations is set up on the flow graph (but as if all edges were reversed). The flow functions on the reverse graph are the (approximate) inverses of the forward flow functions. These equations are then solved using a demand-driven fixed-point-finding procedure.

None of the demand algorithms described above support the ability to answer a query with respect to a user-supplied language of stack configurations. As with previous work on dataflow analysis, those algorithms merge together (via meet, i.e., $\oplus$) the values for each program point, regardless of calling context. In addition, past work on demand-driven dataflow analysis has not examined the issue of providing a witness set of paths to show why the answer to a dataflow query for a particular configuration has the value reported.

The IFDS framework can be extended with the ability to answer a query with respect to a language of stack configurations by applying the reachability algorithms for (unweighted) PDSs [3, 8] on the graphs used in [20, 11]; however, that approach does not work for the more general IDE framework. This paper has shown how to extend the IDE framework to answer a query with respect to a language of stack configurations, using our recent generalization of PDS reachability algorithms to weighted PDSs [25].

It should be noted that, like the algorithms from [22], the algorithm for solving GPR problems given in Section 3 is not guaranteed to terminate for all IDE problems; however, like the algorithms from [22], it does terminate for all copy-constant-propagation problems, all linear-constant-propagation problems, and, in general, all problems for which the set of micro-functions contains no infinite descending chains. The asymptotic cost of the algorithm in this paper is the same as the cost of the demand algorithm for solving IDE problems from [22]; however, that algorithm is strictly less general than the algorithm presented here (cf. Example 4).

An application of the theory of PDSs to interprocedural dataflow analysis has been proposed by Esparza and Knoop [9], who considered several bit-vector problems, as well as the faint-variables problem, which is an IFDS problem [21, Appendix A]. These problems are solved using certain pre^* and $post^*$ queries. With respect to that work, the extension of PDSs to weighted PDSs allows our approach to solve a more general class of dataflow-analysis problems than Esparza and Knoop's techniques can handle; the witness-set generation algorithm can also be used to extend their algorithms. (Esparza and Knoop also consider bit-vector problems for flow-graph systems with parallelism, which we have not addressed.)

Müller-Olm and Seidl have given an interprocedural dataflow-analysis algorithm that determines, for each program point n, the set of all affine relations that hold among program variables whenever n is executed [16]. This method can be re-cast as solving

[5] Logic-programming terminology is not used in [20]; however, the exhaustive algorithm described there has a straightforward implementation as a logic program. A demand algorithm can then be obtained by applying the magic-sets transformation.

a GPR problem (with the same asymptotic complexity). G. Balakrishnan has created a prototype implementation of this method using the WPDS library [24].

Model checking of PDSs has previously been used for verifying security properties of programs [10, 12, 5]. The methods described in this paper should permit more powerful security-verification algorithms to be developed that use weighted PDSs to obtain a broader class of interprocedural dataflow information for use in the verification process.

Acknowledgments

We thank H. Seidl for making available reference [16].

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, New York, NY, 1986. ACM Press.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, volume 1243 of *Lec. Notes in Comp. Sci.*, pages 135–150. Springer-Verlag, 1997.
4. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proc. Symp. on Princ. of Prog. Lang.*, pages 62–73, 2003.
5. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, November 2002.
6. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Symp. on Princ. of Prog. Lang.*, pages 12–25, 2000.
7. E. Duesterwald, R. Gupta, and M.L. Soffa. Demand-driven computation of interprocedural data flow. In *Symp. on Princ. of Prog. Lang.*, pages 37–48, New York, NY, 1995. ACM Press.
8. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. Computer-Aided Verif.*, volume 1855 of *Lec. Notes in Comp. Sci.*, pages 232–247, July 2000.
9. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proceedings of FoSSaCS'99*, volume 1578 of *LNCS*, pages 14–30. Springer, 1999.
10. J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proceedings of TACAS'01*, volume 2031 of *LNCS*, pages 306–339. Springer, 2001.
11. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, New York, NY, October 1995. ACM Press.
12. T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *1999 IEEE Symposium on Security and Privacy*, May 1999.
13. S. Jha and T. Reps. Analysis of SPKI/SDSI certificates using model checking. In *IEEE Comp. Sec. Found. Workshop (CSFW)*. IEEE Computer Society Press, 2002.
14. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct.*, pages 125–140, 1992.
15. U. Moencke and R. Wilhelm. Grammar flow analysis. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lec. Notes in Comp. Sci.*, pages 151–186, Prague, Czechoslovakia, June 1991. Springer-Verlag.
16. M. Müller-Olm and H. Seidl. Computing interprocedurally valid relations in affine programs. Tech. rep., Comp. Sci. Dept., Univ. of Trier, Trier, Ger., January 2003.
17. G. Ramalingam. *Bounded Incremental Computation*, volume 1089 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 1996.

18. T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*. Kluwer Academic Publishers, 1994.
19. T. Reps. Solving demand versions of interprocedural analysis problems. In P. Fritzson, editor, *Proceedings of the Fifth International Conference on Compiler Construction*, volume 786 of *Lec. Notes in Comp. Sci.*, pages 389–403, Edinburgh, Scotland, April 1994. Springer-Verlag.
20. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, New York, NY, 1995. ACM Press.
21. T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Tech. Rep. TR 94-14, Datalogisk Institut, Univ. of Copenhagen, 1994. Available at "http://www.cs.wisc.edu/wpis/papers/diku-tr94-14.ps".
22. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.
23. D. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Symp. on Princ. of Prog. Lang.*, pages 38–48, New York, NY, January 1998. ACM Press.
24. S. Schwoon. WPDS – a library for Weighted Pushdown Systems, 2003. Available from http://www7.in.tum.de/∼schwoon/moped/#wpds.
25. S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Comp. Sec. Found. Workshop*, Wash., DC, 2003. IEEE Comp. Soc.
26. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
27. B. Steffen. Data flow analysis as model checking. In *Int. Conf. on Theor. Aspects of Comp. Softw.*, volume 526 of *Lec. Notes in Comp. Sci.*, pages 346–365. Springer-Verlag, 1991.
28. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Sci. of Comp. Prog.*, 21(2):115–139, 1993.
29. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, March 1992.

A Generation of Witness Sets

Section 3.3 gives an efficient algorithm for determining $\delta(c)$; this section addresses the question of how to obtain $\omega(c)$. It may help to think of this problem as that of examining an infinite graph $\mathcal{G}$ whose nodes are pairs (c, d), where c is a configuration and d a value from D, and in which there is an edge from (c_1, d_1) to (c_2, d_2) labeled with $r \in \Delta$ if and only if $c_1 \xRightarrow{\langle r \rangle} c_2$ and $f(r) \otimes d_2 = d_1$. For a given configuration c, finding $\omega(c)$ means identifying a set of paths $\sigma_1, \ldots, \sigma_k$ such that path σ_i, $1 \leq i \leq k$, leads from some (c, d_i) to some $(c_i, 1)$, where $c_i \in C$, and $\bigoplus_{i=1}^{k} d_i = \delta(c)$. In other words, $\omega(c) = \{\sigma_1, \ldots, \sigma_k\}$ proves that $\delta(c)$ really has the value computed by Algorithm 1. We note the following properties:

- In general, k may be larger than 1, e.g., we might have a situation where $\delta(c) = d_1 \oplus d_2$ because of two paths with values d_1 and d_2, but there may be no single path with value $d_1 \oplus d_2$.
- We want to keep $\omega(c)$ as small as possible. If a witness set contains two paths σ_1 and σ_2, where $v(\sigma_1) \sqsubseteq v(\sigma_2)$, then the same set without σ_2 is still a witness set.

Like $\delta(c)$, $\omega(c)$ will be given indirectly in the form of another annotation (called n) on the transitions of $\mathcal{A}_{pre^*}$. We use two data structures for this, called *wnode* and *wstruc*. If t is a transition, then $n(t)$ holds a reference to a *wnode*. (We shall denote a reference to some entity e by $[e]$.) A *wnode* is a set of *wstruc* items. A *wstruc* item is of the form $(d, [t], [r], N)$ where $d \in D$, $[t]$ is a reference back to t, $r \in \Delta$ is a rule,

Algorithm 2

1 **procedure** *update*(t, r, T)
2 **begin**
3 $\quad \rightarrow := \rightarrow \cup \{t\}$;
4 $\quad d := f(r) \otimes l(T(1)) \otimes \ldots \otimes l(T(|T|))$;
5 $\quad s := (d, [t], [r], (\, n(t') \mid t' \in T\,))$;
6 $\quad$ **if** $l(t) \sqsubseteq d$ **then return**;
7 $\quad$ **if** $n(t) = nil$ or $d \sqsubset l(t)$ **then**
8 $\qquad$ create $n := \{s\}$;
9 $\quad$ **else**
10 $\qquad$ create $n := minimize(S \cup \{s\})$, where $n(t) = [S]$;
11 $\quad n(t) := [n]$;
12 $\quad l(t) := l(t) \oplus d$;
13 $\quad workset := workset \cup \{t\}$
14 **end**

Fig. 8. Modified *update* procedure.

and N contains a sequence of references to *wnode*s. References may be nil, indicating a missing reference.

We can now extend Algorithm 1. The idea is that during execution, if $n(t) = [S]$, then $l(t) = \bigoplus_{(d,[t],[r],N)\in S} d$. An item $(d, [t], [r], N)$ in S denotes the following: Suppose that A_{pre^*} has an accepting path starting with t, and c is the configuration accepted by this path. Then, in the pushdown system, there is a path (or rather, a family of paths) with value d from c to some $c' \in C$, and this path starts with r. An accepting path (in A_{pre^*}) for a successor configuration can be constructed by replacing t with the transitions associated with the *wnode*s in N.

The concrete modifications to Algorithm 1 are as follows: In line 8, set $n \equiv nil$. In line 9, create a *wnode* $n := \{(1, [t], nil, ())\}$ for every $t \in \rightarrow_0$ and set $n(t) := [n]$.

Figure 8 shows a revised *update* procedure. Line 4 of Figure 8 computes the newly discovered value for transition t, and line 5 records how the new path was discovered. In line 6, if $l(t) \sqsubseteq d$, the update will not change $l(t)$ and nothing further needs to be done. If $d \sqsubset l(t)$ (see line 8), the new addition is strictly smaller than any path to t so far, and $n(t)$ only has to reference the new path. If d and $l(t)$ are incomparable, line 10 creates a new set consisting of the previous paths *and* the new path. Even though d is incomparable to $l(t)$, d might approximate ($\sqsubseteq$) one or more elements of S. The procedure *minimize* (not shown) removes these.

It is fairly straightforward to see that the information contained in S allows the reconstruction of a witness set involving t (see above). Moreover, every *wnode* created during execution contains references only to *wnode*s created earlier. Therefore, the process of reconstructing the witness set by decoding *wnode*/*wstruc* information must eventually terminate in a configuration from C.

During execution of the modified algorithm, several *wnode*s for the same transition t can be created; only one of them is referenced by t at any moment, although the other *wnode*s may still be referenced by other transitions. A garbage collector can be used to keep track of the references and remove those nodes to which there is no longer any chain of references from any transition.