Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis [⋆]

Thomas Reps [a,c,∗,1,2], Stefan Schwoon [b,1,3], Somesh Jha [a,1], and David Melski [c,4]

[a] *Computer Sciences Department, University of Wisconsin*

[b] *Institut für Formale Methoden der Informatik, Universität Stuttgart*

[c] *GrammaTech, Inc.*

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.

Key words: interprocedural dataflow analysis, generalized pushdown reachability, meet-over-all-paths problem, weighted pushdown system

⋆ A preliminary version of this paper appeared in the *Proc. of the 10th Int. Static Analysis Symposium*, (San Diego, CA, June 11–13, 2003).

∗ Corresponding author.

Email addresses: `reps@cs.wisc.edu` (Thomas Reps), `schwoosn@informatik.uni-stuttgart.de` (Stefan Schwoon), `jha@cs.wisc.edu` (Somesh Jha), `melski@grammatech.com` (David Melski).

1 Supported by the Office of Naval Research under contract N00014-01-1-0708.

2 Supported by the National Science Foundation under grant CCR-9986308 and by the Alexander von Humboldt Foundation.

3 Supported by EPSRC grant R93346/01.

4 Supported by the Office of Naval Research under contract N00014-03-C-0502, by the Air Force Research Laboratory under contract F30602-02-C-0051, and by DARPA under contract DAAH01-02-C-R120.

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., [1–5]; however, with one exception ([2]), 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 [2], as well as (iii) certain dataflow-analysis problems that cannot be expressed as bit-vector problems, such as linear constant propagation [6] and affine-relation analysis [7]. 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) [8]. 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 both simple constant propagation and 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.

Another theme of the paper is to illustrate a number of classic concepts that arise in interprocedural dataflow analysis (e.g., exhaustive vs. demand evaluation, differential vs. non-differential propagation, etc.) from the viewpoint of the weighted PDS framework.

The algorithms described in the paper have been implemented in two libraries, WPDS [9] and WPDS++ [10], that solve reachability problems on weighted PDSs. These libraries have been used to create prototype implementations of context-sensitive interprocedural dataflow analyses for uninitialized variables, live variables, linear constant propagation, and the detection of affine relationships. WPDS is available on the World Wide Web, and may be used by third parties in the creation of dataflow-analysis tools; WPDS++ will be made available in mid-2005.

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 algorithms for solving GPR problems. Section 4 shows how the GPR framework can be used to solve interprocedural dataflow-analysis problems. Section 5 presents differential algorithms for solving GPR problems. Section 6 discusses related work. Appendices A and B present some technical results that are used in Section 5.

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 subscript* $\mathcal{P}$ *if* $\mathcal{P}$ *is understood. The reflexive transitive closure of* $\Rightarrow$ *is denoted by* $\Rightarrow^*$*. Given a set of configurations* C*, we define* $pre^*(C) \stackrel{\text{def}}{=} \{ c' \mid \exists c \in C\colon c' \Rightarrow^* c \}$ *and* $post^*(C) \stackrel{\text{def}}{=} \{ c' \mid \exists c \in C\colon c \Rightarrow^* c' \}$ *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 [11,12].

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 accepted by some $\mathcal{P}$-automaton. (We frequently omit the prefix $\mathcal{P}$ and simply refer to "automata" if $\mathcal{P}$ is understood.)*

Example 3 *We will use a running example to explain the definitions and algorithms given in Sections 2 and 3. We consider the pushdown system $\mathcal{P}_{ex}$ with control locations p and q, stack alphabet $\{a, b, c, d\}$, and the following five rules:*

$$r_1 = \langle p, a \rangle \hookrightarrow \langle q, b \rangle, \qquad r_2 = \langle p, a \rangle \hookrightarrow \langle p, c \rangle, \qquad r_3 = \langle q, b \rangle \hookrightarrow \langle p, d \rangle,$$
$$r_4 = \langle p, c \rangle \hookrightarrow \langle p, ad \rangle, \qquad r_5 = \langle p, d \rangle \hookrightarrow \langle p, \varepsilon \rangle.$$

Fig. 1(b) shows part of the transition relation $\Rightarrow$ generated by these rules. Fig. 1(a) shows a $\mathcal{P}_{ex}$-automaton (henceforth called $\mathcal{A}_{ex}$) that accepts the set of configurations $C_{ex} = \{\, \langle q, bd^{2k} \rangle \mid k \geq 0 \,\}$. The configurations of C_{ex} are encircled by dotted lines in Fig. 1(b).

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^* [13–15,11] 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:

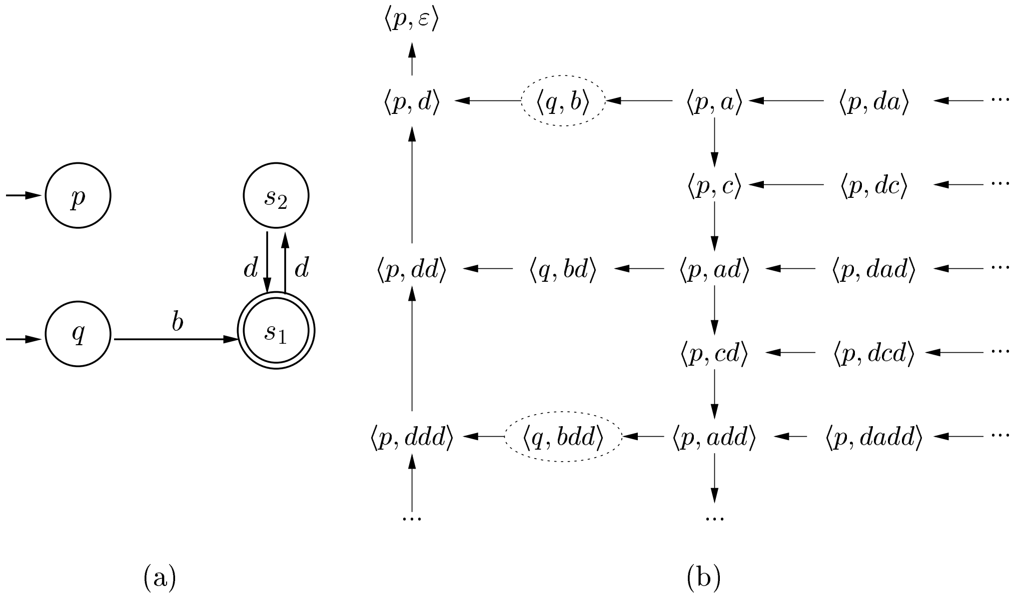


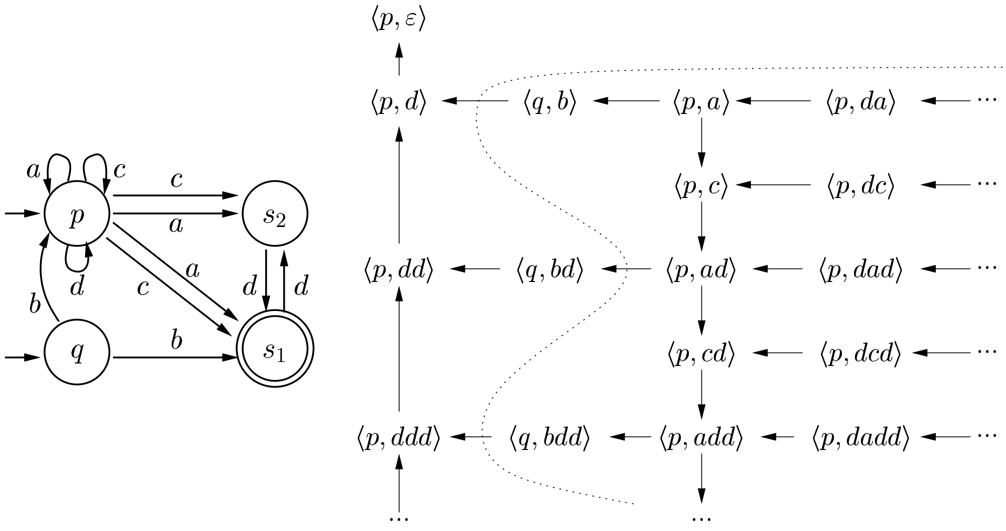
Fig. 1. (a) Automaton $\mathcal{A}_{ex}$. (b) The transition system generated by $\mathcal{P}_{ex}$.

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 [15] an efficient implementation of this procedure is given, which requires $\mathcal{O}(|Q|^2|\Delta|)$ time and $\mathcal{O}(|Q|\,|\Delta| + |{\rightarrow_0}|)$ space.

Example 4 *Applying this procedure to automaton $\mathcal{A}_{ex}$ from Ex. 3 yields the automaton shown in Fig. 2(a), which indeed accepts the set $pre^*(C_{ex})$, including all the configurations to the right of the dotted line in Fig. 2(b).*

An automaton $\mathcal{A}_{post^*}$ accepting $post^*(C)$ can be obtained from a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ by a two-phase procedure. To simplify the presentation, we allow the procedure to compute $\mathcal{A}_{post^*}$ in the form of an automaton with ε-transitions (that is, its transitions are a subset of $Q \times (\Gamma \cup \{\varepsilon\}) \times Q$). However, we assume that the initial automaton $\mathcal{A}$ has no such ε-transitions. The entire procedure is shown below:



(a) (b)

Fig. 2. (a) Automaton accepting $pre^*(C_{ex})$. (b) Extent of $pre^*(C_{ex})$ in $\mathcal{P}_{ex}$.

- **Phase I**
  For each pair $\langle p', \gamma' \rangle$ such that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$, add a new state $q_{p',\gamma'}$.
- **Phase II (saturation phase)**
  In this phase, new transitions are added to the automaton until no more rules can be added. (The symbol $\overset{\gamma}{\rightsquigarrow}$ denotes the relation $(\xrightarrow{\epsilon})^\star \xrightarrow{\gamma} (\xrightarrow{\epsilon})^\star$.)
  The rules for adding new transitions are as follows:
  · If $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, add a transition (p', ϵ, q).
  · If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, add a transition (p', γ', q).
  · If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$ and $p \overset{\gamma}{\rightsquigarrow} q$ in the current automaton, first add $(p', \gamma', q_{p',\gamma'})$ and then $(q_{p',\gamma'}, \gamma'', q)$.

$\mathcal{A}_{post^*}$ can be constructed in time and space $\mathcal{O}(n_P n_\Delta (n_1 + n_2) + n_P n_0)$, where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, $n_0 = |{\to_0}|$, $n_1 = |Q \backslash P|$, and n_2 is the number of different pairs (p', γ') such that there is a rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ in Δ [11].

In Sections 3 and 5, we develop generalizations of these procedures.

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 Defn. 5.

Definition 5 *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.*

Defn. 5(1) and Defn. 5(5) mean that $(D, \oplus)$ is a **meet semilattice with no infinite descending chains**.

Definition 6 *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* $f \colon \Delta \to D$ *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 $v(\sigma) \stackrel{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. 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., $c \xRightarrow{\langle r_1 \rangle} \cdots \xRightarrow{\langle r_k \rangle} c'$.

We now define two kinds of **generalized pushdown reachability (GPR) problems**:

Definition 7 *Let* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *be a weighted pushdown system, where* $\mathcal{P} = (P, \Gamma, \Delta)$, *and let* $C \subseteq P \times \Gamma^*$ *be a regular set of configurations. The* **generalized pushdown predecessor (GPP) problem** *is to find for each* $c \in P \times \Gamma^*$:

- $\delta(c) \stackrel{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in path(c, c'), c' \in C \,\}$;
- *a* **witness set** *of paths* $\omega(c) \subseteq \bigcup\limits_{c' \in C} path(c, c')$ *such that* $\bigoplus\limits_{\sigma \in \omega(c)} v(\sigma) = \delta(c)$.

The **generalized pushdown successor (GPS) problem** *is to find for each* $c \in P \times \Gamma^*$*:*

- $\delta(c) \stackrel{\text{def}}{=} \bigoplus\{\, v(\sigma) \mid \sigma \in path(c', c), c' \in C \,\}$*;*
- *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 GPP and GPS problems, because of Defn. 5(5) (i.e., "no infinite descending chains"), for each $c \in P \times \Gamma^*$ it is always possible to identify a witness set $\omega(c)$ that is finite.

3 Solving Generalized Pushdown Reachability Problems

Throughout this 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.

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

Because GPR problems concern infinite graphs, and not just an infinite set of paths, they differ from other work on meet-over-all-paths problems. As in ordinary pushdown-reachability problems [13–15], the infinite nature of GPR problems is addressed by reporting the answer in an indirect fashion, namely, in the form of an (annotated) automaton.

Answer automata without their annotations are identical to the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata created by the algorithms of [15,11]. The annotations allow us to obtain $\delta(c)$ and $\omega(c)$ values. For instance, as described in Section 3.1, 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 created by the algorithm for solving GPP problems; for $c \notin pre^*(C)$, the values of $\delta(c)$ and $\omega(c)$ are 0 and $\emptyset$, respectively. (A similar statement can be made for $c \in post^*(C)$ and the automaton created by the algorithm for solving GPS problems; see Section 3.2.)

This section presents the algorithm from [8] for solving GPP problems. The algorithm is presented in several stages:

- We first define a context-free grammar that characterizes certain 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 GPP problem. We use the grammar characterization of transition sequences, together with previously known results on a certain kind of grammar-valuation problem [16,17], to derive an algorithm for solving GPP problems.
- However, the initial solution 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 [15,11].

3.1.1 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 *pre** *witnesses* for $c \in P \times \Gamma^*$ with respect to C: $\mathit{PreStarWitnesses}(c, C) = \bigcup_{c' \in C} \mathit{path}(c, c')$.

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 $\rightarrow_0$.

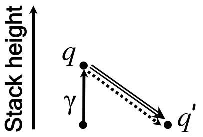
We say that a configuration $c = \langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$ is *accepted* by $\mathcal{PA}$ if there is a path to a configuration $\langle q_f, \epsilon\rangle$ such that $q_f \in F$. Note that because $\mathcal{A}$ has no transitions leading to initial states, $\mathcal{PA}$'s 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 $(Q \setminus P)$, 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 $\mathcal{P}$, after $\mathcal{P}$ was started in configuration c. In other words, $\mathcal{PA}$ accepts a configuration c iff $c \in \mathit{pre}^*(C)$.

The first language that we define characterizes the *pop sequences* of $\mathcal{PA}$.

Definition 8 (Pop Sequence) A *pop sequence* for $q \in Q$, $\gamma \in \Gamma$, and $q' \in Q$ is a sequence of transitions of $\mathcal{PA}$'s transition relation that (i) starts in a

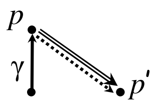
Production	for each
(1) $PopSeq_{(q,\gamma,q')} \rightarrow \varepsilon$	$q \xrightarrow{\gamma} q' \in \rightarrow_0$
(2) $PopSeq_{(p,\gamma,p')} \rightarrow \varepsilon$	$\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$
(3) $PopSeq_{(p,\gamma,q)} \rightarrow PopSeq_{(p',\gamma',q)}$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

Fig. 3. A context-free grammar for the pop sequences of $\mathcal{PA}$, and the $\mathcal{PA}$ rules that correspond to each production.



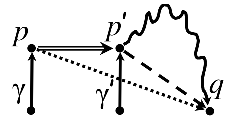
(1) $PopSeq_{(q,\gamma,q')} \rightarrow \varepsilon$

$q \xrightarrow{\gamma} q' \in \rightarrow_0$



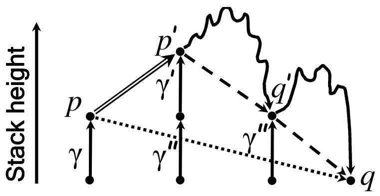
(2) $PopSeq_{(p,\gamma,p')} \rightarrow \varepsilon$

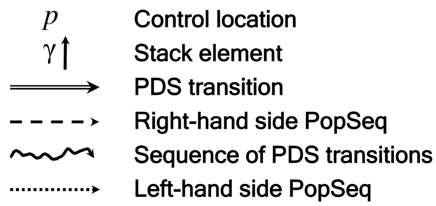
$\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$



(3) $PopSeq_{(p,\gamma,q)} \rightarrow PopSeq_{(p',\gamma',q)}$

$\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta, q \in Q$





(4) $PopSeq_{(p,\gamma,q)} \rightarrow PopSeq_{(p',\gamma',q')}\ \ PopSeq_{(q',\gamma'',q)}$

$\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta, q, q' \in Q$

Fig. 4. Schematic diagram of the types of transition sequences captured by the *PopSeq* productions from Fig. 3.

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 string $w' \in \Gamma^+$, except in the last step, when the stack shrinks to w.

Note that, in general, there are many similar pop sequences that differ only in the untouched part of the stack (i.e., w). Moreover, for all w, there is a pop sequence for q, γ, and q' with untouched stack w if and only if there is a pop sequence for q, γ, and q' with (untouched) stack ε.

The family of pop sequences for a given q, γ, and q' can be characterized by the complete derivation trees[5] derived from nonterminal $PopSeq_{(q,\gamma,q')}$, using the context-free grammar shown in Fig. 3. Fig. 4 depicts the types of transition sequences captured by the *PopSeq* productions from Fig. 3.

Example 9 *Recall the pushdown system $\mathcal{P}_{ex}$ from Ex. 3. Its transition system*

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

(cf. Fig. 1(b)) admits the sequence

$$\langle p, a\rangle \xRightarrow{\langle r_2\rangle} \langle p, c\rangle \xRightarrow{\langle r_4\rangle} \langle p, ad\rangle \xRightarrow{\langle r_1\rangle} \langle q, bd\rangle \xRightarrow{\langle r_3\rangle} \langle p, dd\rangle \xRightarrow{\langle r_5\rangle} \langle p, d\rangle \xRightarrow{\langle r_5\rangle} \langle p, \varepsilon\rangle,$$

which is a pop sequence for p, a, and p in which the untouched part of the stack is empty. Fig. 5 shows how this sequence is captured by a complete derivation tree of the grammar that corresponds to $\mathcal{PA}_{ex}$—the combined PDS created from $\mathcal{P}_{ex}$ and $\mathcal{A}_{ex}$. (In this example, the pop-sequence derivation tree only makes use of grammar rules that correspond to PDS rules of $\mathcal{P}_{ex}$.) The rule sequence that makes up the pop sequence is obtained from a preorder listing of the tree: subsequence $[r_2r_4]$ corresponds to the part of the tree up to the branching, $[r_1r_3r_5]$ to the left branch, and $[r_5]$ to the right branch. Note that the left branch is itself another pop sequence for p, a, and p.

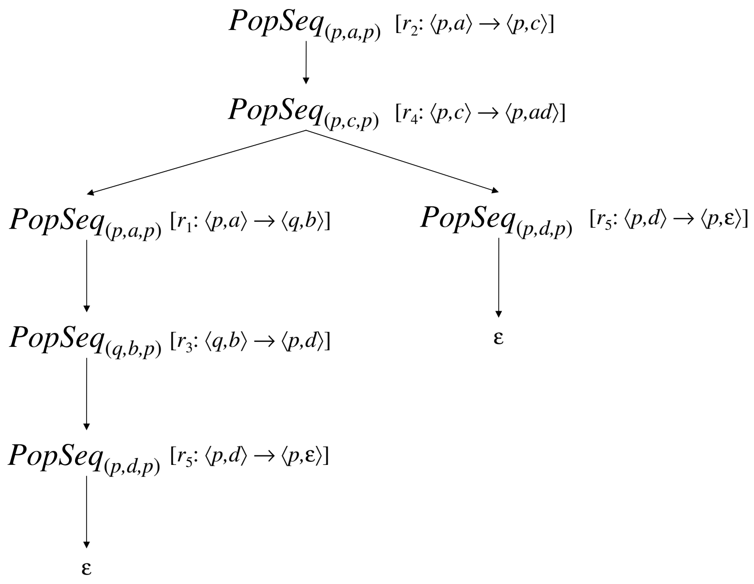


Fig. 5. A complete derivation tree for $PopSeq_{(p,a,p)}$.

Theorem 10 *PDS $\mathcal{PA}$ has a pop sequence for q, γ, and q' iff nonterminal $PopSeq_{(q,\gamma,q')}$ of the grammar shown in Fig. 3 has a complete derivation tree. Moreover, for each complete derivation tree with root $PopSeq_{(q,\gamma,q')}$, a preorder listing of the derivation tree's production instances (where Fig. 3 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 $PopSeq_{(q,\gamma,q')}$.*

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 (see Fig. 4).

The left-hand and right-hand sides of the productions in Fig. 3 reflect the pop-sequence obligations incurred by the corresponding rule of $\mathcal{PA}$. $\square$

To capture the set $\mathit{PreStarWitnesses}(\langle p, \gamma_1\gamma_2 \dots \gamma_n\rangle, C)$, where C is recognized by automaton $\mathcal{A}$, we extend the context-free grammar from Fig. 3 by the set of productions

$$\begin{aligned}
\mathit{Accepting}[\gamma_1\gamma_2 \dots \gamma_n]_{(p,q)} &\rightarrow \mathit{PopSeq}_{(p,\gamma_1,q_1)} \;\; \mathit{PopSeq}_{(q_1,\gamma_2,q_2)} \;\; \dots \;\; \mathit{PopSeq}_{(q_{n-1},\gamma_n,q)} \\
&\qquad \text{for each } p \in P, q_i \in Q, \text{ for } 1 \leq i \leq n-1; \text{ and } q \in F \\
\mathit{Accepted}[\gamma_1\gamma_2 \dots \gamma_n]_{(p)} &\rightarrow \mathit{Accepting}[\gamma_1\gamma_2 \dots \gamma_n]_{(p,q)} \qquad \text{for each } p \in P, q \in F
\end{aligned}$$

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

3.1.2 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 $\langle q, \gamma\rangle \hookrightarrow \langle q', \epsilon\rangle$ 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 11 *[16,17] Let $(S, \sqcap)$ be a meet semilattice. An* **abstract grammar** *over $(S, \sqcap)$ is a collection of context-free grammar productions, where each production θ has the form*

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

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*

Production	for each
(1) $PopSeq_{(q,\gamma,q')} \to g_1(\epsilon)$	$(q,\gamma,q') \in \to_0$
$g_1 = 1$	
(2) $PopSeq_{(p,\gamma,p')} \to g_2(\epsilon)$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$
$g_2 = f(r)$	
(3) $PopSeq_{(p,\gamma,q)} \to g_3(PopSeq_{(p',\gamma',q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
$g_3 = \lambda x.f(r) \otimes x$	
(4) $PopSeq_{(p,\gamma,q)} \to g_4(PopSeq_{(p',\gamma',q')}, PopSeq_{(q',\gamma'',q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q, q' \in Q$
$g_4 = \lambda x.\lambda y.f(r) \otimes x \otimes y$	
(5) $Accepting[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)} \to g_5(PopSeq_{(p,\gamma_1,q_1)}, PopSeq_{(q_1,\gamma_2,q_2)}, \ldots, PopSeq_{(q_{n-1},\gamma_n,q)})$	$p \in P, q_i \in Q,$ for $1 \le i \le n-1,$ and $q \in F$
$g_5 = \lambda x_1.\lambda x_2 \ldots \lambda x_n.x_1 \otimes x_2 \otimes \ldots \otimes x_n$	
(6) $Accepted[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)} \to g_6(Accepting[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)})$	$p \in P, q \in F$
$g_6 = \lambda x.x$	

Fig. 6. An abstract grammar problem for the weighted *pre** problem.

nonterminal X, the value

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

The value $m_G(X)$ is called the **meet-over-all-derivations** *value for nonterminal X.*

Because the complete derivation trees with root $Accepted[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)}$ encode the transition sequences by which $\mathcal{WA}$ accepts $\langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$, to cast a GPP problem 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 Fig. 6: 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 10).

Example 12 *Consider once again the pushdown system $\mathcal{P}_{ex}$ introduced in Ex. 3 and suppose that we assign a non-negative integer weight to each rule. Let $\mathcal{S}_D = (\mathbb{N}_0, \min, +, \infty, 0)$ be the* **Dijkstra semiring***, whose domain is the non-negative integers, and in which values along a path are added up. Moreover, let $\mathcal{W}_{ex} = (\mathcal{P}_{ex}, \mathcal{S}_D, f)$ be a weighted pushdown system. For the* *purpose of our example, let*

$$f(r_1) = 5, \qquad f(r_2) = 4, \qquad f(r_3) = 3, \qquad f(r_4) = 2, \qquad f(r_5) = 1.$$

The transition relation of $\mathcal{W}_{ex}$, complete with the weights given by f, is shown in Fig. 7.

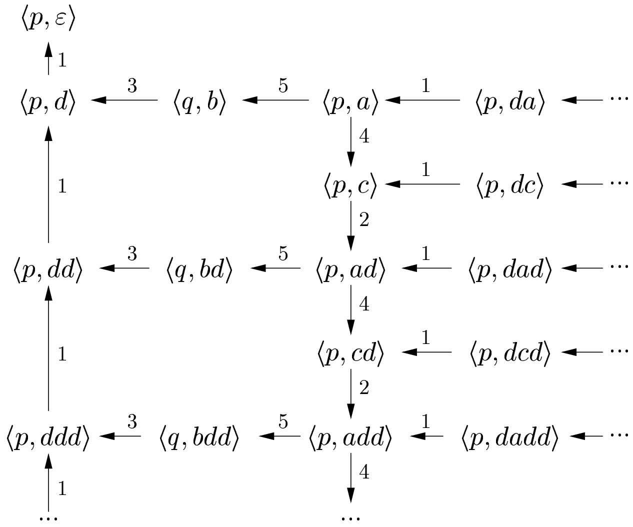


Fig. 7. Weighted transition system of $\mathcal{W}_{ex}$.

Fig. 8 shows the weighted version of the derivation tree from Fig. 5, where the functions associated with the productions are taken from Fig. 6. In each function, the λ-operators capture the values obtained from the children of the associated node, from left to right. Thus, the value obtained at the root is 16, *which is indeed the value of the pop sequence represented by the tree (cf. Fig. 7).*

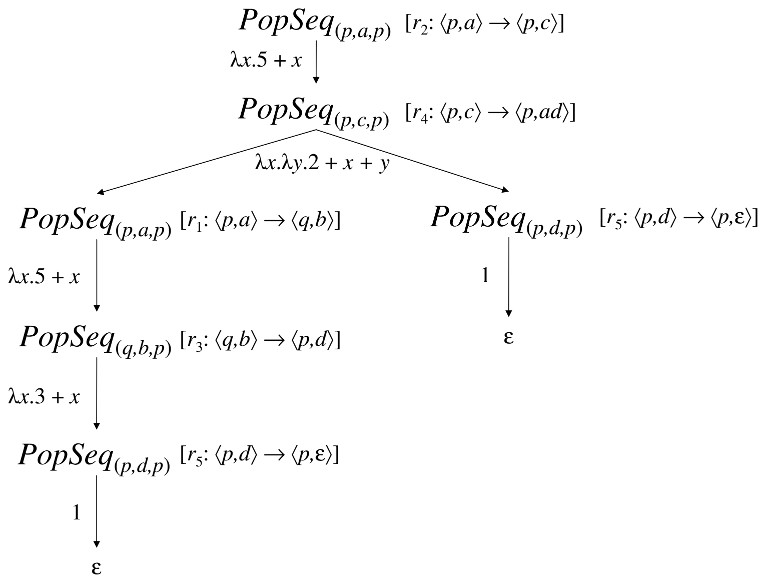


Fig. 8. Complete derivation tree for $PopSeq_{(p,a,p)}$ with weights from Fig. 6.

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

Theorem 13 *[16,17] 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 meet semilattice $(S, \sqcap)$ has no infinite descending chains.
(2) Every production function g_θ in G is distributive, i.e.,

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

for arbitrary, non-empty, finite index sets $I_1, \ldots, I_k$.
(3) Every production function g_θ in G is strict in $\top$ in each argument, where $\top$ is the greatest element of $(S, \sqcap)$.

The abstract grammar problem given in Fig. 6 meets the conditions of Theorem 13 because

(1) By Defn. 5(1), the $\oplus$ operator is associative, commutative, and idempotent; hence, $(D, \oplus)$ is a meet semilattice. By Defn. 5(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 Defn. 5(3).
(3) By Defn. 5(1), $\oplus$ has the identity element 0; hence, $(D, \oplus)$ is a meet semilattice with greatest element 0. Production functions $g_3, \ldots, g_6$ are strict in 0 in each argument because 0 is an annihilator with respect to $\otimes$ (Defn. 5(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 GPP 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 Fig. 6.
- 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.

3.1.3 A More Efficient Algorithm for the GPP 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 Fig. 6 with left-hand-side nonterminal $\mathit{Accepting}[\gamma_1\gamma_2 \ldots \gamma_n]_{(p,q)}$. However, we can improve on the algorithm of the previous section because not all instantiations of the productions listed in Fig. 6 are relevant to the final solution; we want to prevent the algorithm from exploring useless nonterminals of the grammar shown in Fig. 6.

Moreover, all GPP questions with respect to a given target-configuration set C involve the same subgrammar for the *PopSeq* nonterminals. As in the (ordinary) pushdown-reachability problem [13–15], the information about whether a complete derivation tree with root nonterminal $\mathit{PopSeq}_{(q,\gamma,q')}$ exists (i.e., whether $\mathit{PopSeq}_{(q,\gamma,q')}$ 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 *PopSeq* subgrammar lazily saves us from having to construct the entire *PopSeq* subgrammar. Productive nonterminals represent automaton transitions, and the productions that involve any given transition can be constructed on-the-fly, as is done in Alg. 1, shown in Fig. 9.

Fig. 9 presents an algorithm for creating a weighted automaton for the GPP problem. In essence, the algorithm does the following: it starts with the automaton $\mathcal{A}$, which accepts the set of configurations C. Each transition t of the automaton is labeled with an element from the semiring $\mathcal{S}$ (denoted by $l(t)$). Initially, all of the transitions in $\mathcal{A}$ are labeled with 1. We add transitions to $\mathcal{A}$ according to the following saturation rule:

If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ and there is a path for string w from p' to q with cost c in the current automaton, either (i) introduce a transition (p, γ, q) if the automaton does not already contain such a transition, or (ii) change the label on (p, γ, q) if (p, γ, q) already occurs in the automaton. The label of transition (p, γ, q) is computed as follows:

$$\begin{array}{ll} f(r) \otimes c & \text{if } (p, \gamma, q) \text{ is a new transition} \\ (f(r) \otimes c) \;\oplus\; l(p, \gamma, q) & \text{otherwise} \end{array}$$

The cost of a path in the automaton is computed by taking the $\otimes$ of the labels on the transitions along the path.

It is relatively straightforward to see that Alg. 1 solves the grammar problem for the *PopSeq* subgrammar from Fig. 6: *workset* contains the set of transitions (*PopSeq* nonterminals) whose value $l(t)$ has been updated since it was last

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 $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ that accepts C,
such that $\mathcal{A}$ has no transitions into P states.

Output: a $\mathcal{P}$-automaton $\mathcal{A}_{pre^*} = (Q, \Gamma, \rightarrow, P, F)$ that accepts $pre^*(C)$;
a function l that maps every $(q, \gamma, q') \in \rightarrow$ to the value of
$m_G(PopSeq_{(q,\gamma,q')})$ in the abstract grammar problem defined in Fig. 6.

1 **procedure** *update*(t, v)
2 **begin**
3 $\quad \rightarrow := \rightarrow \cup \{t\}$
4 $\quad newValue := l(t) \oplus v$
5 $\quad$ **if** $newValue \neq l(t)$ **then**
6 $\qquad workset := workset \cup \{t\}$
7 $\qquad l(t) := newValue$
8 **end**
9
10 $\rightarrow := \rightarrow_0$; $workset := \rightarrow_0$; $l := \lambda t.0$
11 **for all** $t \in \rightarrow_0$ **do** $l(t) := 1$
12 **for all** $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $update((p, \gamma, p'), f(r))$
13 **while** $workset \neq \emptyset$ **do**
14 $\quad$ select and remove a transition $t = (q, \gamma, q')$ from $workset$
15 $\quad$ **for all** $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in \Delta$ **do** $update((p_1, \gamma_1, q'), f(r) \otimes l(t))$
16 $\quad$ **for all** $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta$ **do**
17 $\qquad$ **for all** $t' = (q', \gamma_2, q'') \in \rightarrow$ **do** $update((p_1, \gamma_1, q''), f(r) \otimes l(t) \otimes l(t'))$
18 $\quad$ **for all** $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \gamma_2\gamma \rangle \in \Delta$ **do**
19 $\qquad$ **if** $t' = (p', \gamma_2, q) \in \rightarrow$ **then** $update((p_1, \gamma_1, q'), f(r) \otimes l(t') \otimes l(t))$
20 **return** $((Q, \Gamma, \rightarrow, P, F), l)$

Fig. 9. An algorithm for creating a weighted automaton for the weighted pre^* problem.

considered; in line 10 all values are set to 0. Lines 11–12 process the rules of types (1) and (2), respectively. Lines 13–19 represent the fixed-point-finding loop: lines 15, 17, and 19 simulate the processing of rules of types (3) and (4) that involve transition t on their right-hand side. A function call $update(t, v)$ computes the new value for transition t in terms of $l(t)$ and v. Note that line 7 can change $l(t)$ only to a smaller value (with respect to $\sqsubseteq$). The iterations continue until the values of all transitions stabilize, i.e., *workset* is empty.

From the observation that Alg. 1 is simply a different way of expressing the grammar problem for the *PopSeq* 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 [15]—the only major difference being that transitions are stored in a workset and processed multiple times, whereas in [15] each transition is processed exactly once. Thus, the time complexity increases from the $\mathcal{O}(|Q|^2|\Delta|)$

complexity of the unweighted case [15] by a factor that is no more than the length of the maximal-length descending chain to any value that appears in the annotated automaton.

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. [18, Algorithm 3.4]).

Example 14 *Recall the weighted pushdown system $\mathcal{W}_{ex}$ introduced in Ex. 12. A GPP problem for $\mathcal{W}_{ex}$ formulates a multi-target shortest-path problem on its infinite transition system, where the targets are some regular set of configurations (say, C_{ex}, see Ex. 3): In the automaton computed by Alg. 1, each accepting path for some configuration c corresponds to one or more pre^* witnesses for c with respect to C_{ex}; using minimum as the combiner ensures that the value of the shortest path is retained. Fig. 10(a) shows the initial weighted automaton accepting C_{ex}, in which all transitions are labeled with the* 1*-element of the semiring (which in this example is the number* 0*). Applying the saturation rule to this automaton leads to the following actions:*

- *First, we have $r_5 = \langle p, d\rangle \hookrightarrow \langle p, \varepsilon\rangle$, and $p \xrightarrow{\varepsilon}{}^* p$ with weight* 0 *holds trivially. Therefore, we add $p \xrightarrow{d} p$ with weight $f(r_5) = 1$.*
- *Next, we can consider the rule $r_1 = \langle p, a\rangle \hookrightarrow \langle q, b\rangle$ and the path $q \xrightarrow{b}{}^* s_1$ with weight* 0*, which allows us to add a new transition $p \xrightarrow{a} s_1$ with weight $f(r_1) + 0 = 5$.*
- *This addition creates a path $p \xrightarrow{ad}{}^* s_2$ with weight $5 + 0$. Because we have $r_4 = \langle p, c\rangle \hookrightarrow \langle p, ad\rangle$ and $f(r_4) = 2$, the next addition is $p \xrightarrow{c} s_2$ with weight* 7*.*
- *Similar considerations lead to $p \xrightarrow{a} s_2$ with weight $f(r_2) + 7 = 11$, $p \xrightarrow{c} s_1$ with $f(r_4) + 11 = 13$, $q \xrightarrow{b} p$ with $f(r_3) + 1 = 4$, $p \xrightarrow{a} p$ with $f(r_1) + 4 = 9$, and $p \xrightarrow{c} p$ with $f(r_4) + 9 + 1 = 12$.*
- *At this point, the saturation procedure reaches a fixed point—i.e., the automaton shown in Fig. 10(b). For instance, we could still consider rule $r_2 = \langle p, a\rangle \hookrightarrow \langle p, c\rangle$ and $p \xrightarrow{c}{}^* s_1$, which would contribute the value $f(r_2) + l(p, c, s_1) = 4 + 13$ to the weight of $p \xrightarrow{a} s_1$. However, because we already have $l(p, a, s_1) = 5$ and $\min\{5, 17\} = 5$, this would not make a difference.*

The automaton produced by this procedure allows us to determine, for each configuration $c \in pre^*(C_{ex})$, the length of the shortest path from c to some configuration in C_{ex}. For instance, looking at $c_{ex} := \langle p, dc\rangle$ we find the accepting path $p \xrightarrow{d} p \xrightarrow{c} s_1$, whose value is $1+13 = 14$. In fact, the shortest path in $\mathcal{W}_{ex}$ from c_{ex} to a configuration in C_{ex} is $\sigma = [r_5 r_4 r_2 r_4 r_1]$ leading to $\langle q, bdd\rangle$,

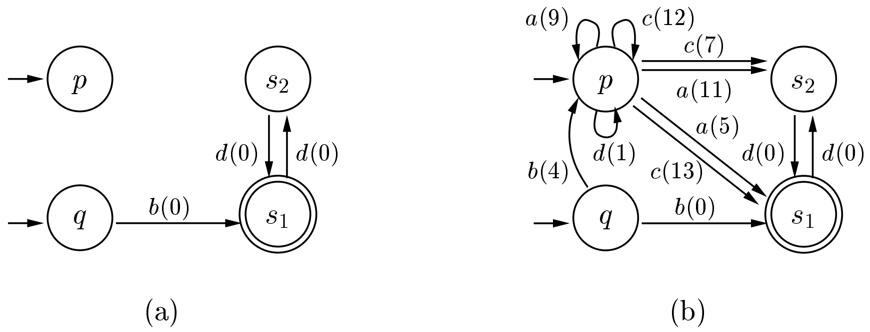


Fig. 10. (a) Initial weighted automaton for C_{ex}. (b) Automaton after applying Alg. 1.

and $v(\sigma) = 1 + 2 + 4 + 2 + 5 = 14$.

Please note that the Dijkstra semiring is a particularly simple example for the GPP framework in the following sense:

- Using min as the combiner operation imposes a total ordering on the semiring domain. In general, the GPP framework can deal with partial orderings, and the values obtained for each configuration can stem from the combination of multiple paths. In this sense, the value of 14 for c_{ex} can be seen as a *summary* of all the paths leading to C_{ex}, the summary in this case being simply the value of the shortest path.
- The extender operator, $+$, is commutative. In general, this is not required, and the order of arguments to $\otimes$ in Alg. 1 really matters.

Section 4 will show some examples that do not exhibit these characteristics.

Alg. 1 is a dynamic-programming algorithm for determining $\delta(c)$; Section 3.1.4 describes how to extend Alg. 1 to keep additional annotations on transitions so that a path set $\omega(c)$ can be obtained.

3.1.4 Generation of Witness Sets

Section 3.1.3 gives an efficient algorithm for determining $\delta(c)$; this section addresses the question of how to obtain a $\omega(c)$ set that is finite. For a given configuration c, finding $\omega(c)$ means identifying a set of paths $\pi_1, \ldots, \pi_k$ in the transition relation of the weighted PDS such that, for $1 \leq i \leq k$, each path π_i leads from c to some $c_i \in C$, $v(\pi_i) = d_i$, and $\bigoplus_{i=1}^{k} d_i = \delta(c)$. 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 π_1 and π_2, where $v(\pi_1) \sqsubseteq v(\pi_2)$, then the same set without π_2 is still a witness set.

As with many dynamic-programming problems, a "global reason" for an answer can be obtained by recording "local reasons". In this case, to obtain a finite witness set $\omega(c)$, we will create a directed hypergraph $\mathcal{G}_{pop} = (N, E)$, where $N \subseteq (\rightarrow \times D)$ and $E \subseteq (N \times \Delta \times N^*)$.[6] A node $n = (t, d) \in N$, where $t = (p, \gamma, q)$, records that there exist pop sequences $\sigma_1, \ldots, \sigma_k$ for p, γ, q and $d = \bigoplus_{i=1}^{k} d_i$, where $d_1, \ldots, d_k$ are the semiring values accumulated along these paths:

- If $t \in \rightarrow_0$ is a transition from $\mathcal{A}$ and $d = 1$, then $k = 1$ and $\sigma_1 = \varepsilon$; this fact is represented by a node $(t, 1)$ that has no incoming hyperedges.
- For a node $n = (t, d)$ for which $t \notin \rightarrow_0$ is not a transition from $\mathcal{A}$, each hyperedge $(n, r, n_1 \ldots n_m)$ corresponds to a collection of pop sequences for p, γ, q; each of these pop sequences is of the form $r\tau_1 \ldots \tau_m$, where each τ_i, for $1 \le i \le m$, is a pop sequence for n_i.

Once $\mathcal{G}_{pop}$ is constructed, the information in it captures a witness set for any given configuration c: if $t_1 \ldots t_m$ is a path in $\mathcal{A}_{pre^*}$ by which c is accepted, then $\omega(c)$ consists of every sequence $\sigma_1 \ldots \sigma_m$, where, for every $1 \le i \le m$, σ_i is a pop sequence for $(t_i, l(t_i))$. (It should be noted that $\mathcal{G}_{pop}$ is a succinct representation of $\omega(c)$; in the worst case, (i) the length of a path in $\omega(c)$ can be exponential in the size of $\mathcal{G}_{pop}$, and (ii) the cardinality of $\omega(c)$ can be doubly exponential in the size of $\mathcal{G}_{pop}$. Thus, in some cases it may be important for witness sets to be reported as hypergraphs.)

Example 15 *Fig. 11 shows what $\mathcal{G}_{pop}$ looks like for the automaton created in Ex. 14. Hyperedges are shown as a collection of simple edges with the label inside a box. Notice that the hyperedge labeled with r_5 has an empty sequence of nodes as its source because it is derived from the rule $r_5 = \langle p, d\rangle \hookrightarrow \langle p, \varepsilon\rangle$.*

In Ex. 14, the configuration $c_{ex} = \langle p, dc\rangle$ is accepted by the path $p \xrightarrow{d} p \xrightarrow{c} s_1$. Therefore, $\omega(c_{ex})$ consists of the pop sequence for $((p, d, p), 1)$, followed by the sequence for $((p, c, s_1), 13)$. (Because the semiring in this example is totally ordered, we end up with just one pop sequence per transition.) The former is merely $[r_5]$; the latter is $[r_4 r_2 r_4 r_1]$. In fact, these sequences together form exactly the shortest path mentioned in Ex. 14.

To implement the idea outlined above, we extend Alg. 1 as follows: In line 10, the empty hypergraph is created by setting $N := \emptyset$ and $E := \emptyset$. In line 11, a

[6] Each hyperedge is of the form $(n, r, n_1 \ldots n_m)$; n is the *target* of the hyperedge; $n_1, \ldots, n_m$ are its (ordered) *sources*. The order of source nodes matters; i.e., two hyperedges that have the same source nodes in different orders are different hyperedges.

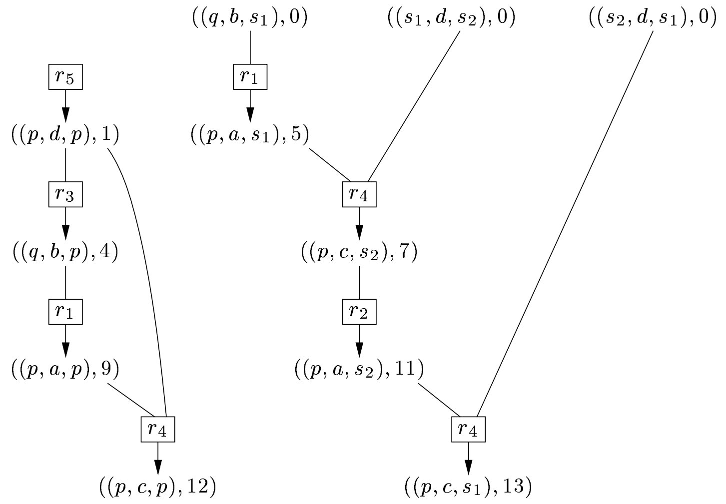


Fig. 11. The graph $\mathcal{G}_{pop}$ for the automaton from Fig. 10(b).

Algorithm 2

procedure *update*(t, v, r, T)
begin
  $\rightarrow := \rightarrow \cup \{t\}$
  $newValue := l(t) \oplus v$
  if $newValue = l(t)$ **then return**
  $workset := workset \cup \{t\}$
  $N := N \cup \{(t, newValue)\}$
  // Record the contribution of v to $newValue$
  $E := E \cup \{((t, newValue), r, (t_1, l(t_1)) \ldots (t_m, l(t_m)))\}$ where $t_1 \ldots t_m = T$
  // Copy hyperedges whose values are not subsumed by v
  for all $((t, l(t)), r', (t_1, d_1) \ldots (t_m, d_m)) \in E$ **do**
    if $v \not\sqsubseteq f(r') \otimes \bigotimes_{i=1}^{m} d_i$ **then**
      $E := E \cup \{((t, newValue), r', (t_1, d_1) \ldots (t_m, d_m))\}$
  $l(t) := newValue$
end

Fig. 12. Modified *update* procedure.

node $(t, 1)$ is added to N for every $t \in \rightarrow_0$. Moreover, we replace the *update* procedure by the revised version shown in Fig. 12.

In Fig. 9, *update* takes two arguments: t and v. To create the witness structure $\mathcal{G}_{pop}$, the *update* procedure given in Fig. 12 needs access to r, the rule used, and T, the list of transitions used; hence, *update* now takes four arguments: t, v, r, and T. The four calls on *update* in Fig. 9 are modified as follows:

line 12: $\quad update((p, \gamma, p'), f(r), r, \varepsilon)$

line 15: $\quad update((p_1, \gamma_1, q'), f(r) \otimes l(t), r, t)$

line 17: $\quad update((p_1, \gamma_1, q''), f(r) \otimes l(t) \otimes l(t'), r, t\,t')$

line 19: $\quad update((p_1, \gamma_1, q'), f(r) \otimes l(t') \otimes l(t), r, t'\,t)$

In line 5 of Fig. 12, if $newValue = l(t)$, then the update does not change $l(t)$, and nothing further needs to be done. In line 7, the node $(t, newValue)$ is added to $\mathcal{G}_{pop}$, and in lines 9–13 hyperedges to the new node are added.

- In line 9, the addition of $((t, newValue), r, (t_1, l(t_1)) \ldots (t_m, l(t_m)))$ records the contribution of r and T to *newValue*. That is, r and T contributed value v, which is supported by the witness information available in the hyperedges that have target $(t_i, l(t_i))$, $1 \leq i \leq m$.
- In addition, in lines 11–13 copies of the hyperedges that have target $(t, l(t))$ are created, but now with target $(t, newValue)$. The check in line 12 assures that such hyperedges are added only if the values accumulated along the corresponding pop sequences actually contribute to *newValue*.

In an implementation, one would also want to keep the hypergraph as small as possible, which can be accomplished by garbage collecting the parts of $\mathcal{G}_{pop}$ that cannot affect any node $(t, l(t))$, where $t \in \rightarrow$ and $l(t)$ is the current value associated with t. Note that hyperedges created during *update* contain only references to nodes created strictly earlier, and thus $\mathcal{G}_{pop}$ cannot contain cycles. If each target node holds a reference to each of its incoming hyperedges, and each hyperedge holds a reference to each of its source nodes, reference counting can be used to identify the nodes and hyperedges that can be collected.

3.2 Solving Generalized Pushdown Successor Problems

This section presents an algorithm for solving GPS problems. Given a weighted pushdown system $\mathcal{P}$ and a $\mathcal{P}$-automaton $\mathcal{A}$ that recognizes a set of configurations C, the algorithm creates an annotated $\mathcal{P}$-automaton (with ε-transitions, cf. Section 2.1) that (i) recognizes $post^*(C)$, and (ii) for each $c \in post^*(C)$, provides a way to read out the values of $\delta(c)$ and $\omega(c)$. Without loss of generality, we assume that (i) $\mathcal{A}$ itself contains no ε-transitions, and (ii) $\mathcal{A}$ has no transitions into P states.

The presentation in this section parallels that of Section 3.1; the algorithm for solving GPS problems will be presented in several stages:

- We first define a context-free grammar that characterizes certain sequences of transitions that can be made by a pushdown system $\mathcal{P}$ and an automaton

$\mathcal{A}$ for C.

- We use the grammar characterization of transition sequences to derive an algorithm for solving GPS problems.
- Again, to improve the performance we specialize the computation, ending up with an algorithm for creating an annotated automaton that is quite similar to the *post** algorithm from [15,11].

Similar to what was done in Section 3.1, the first step is to make some definitions that aid in reasoning about the set of paths that lead from configurations in a regular set C to a configuration c. We call this set the *post* witnesses* for $c \in P \times \Gamma^*$ with respect to C: $PostStarWitnesses(c, C) = \bigcup_{c' \in C} path(c', c)$.

Again, it is convenient to combine $\mathcal{P}$ and $\mathcal{A}$ in sequence to create a combined PDS, which we will call $\mathcal{A}^R\mathcal{P}$; however, here the transitions for $\mathcal{A}$ are reversed, and the reversed automaton's rules will precede those of $\mathcal{P}$. As a prelude to the construction of $\mathcal{A}^R\mathcal{P}$, first consider the combination of $\mathcal{P}$ and $\mathcal{A}$ defined as follows:

> The states are $P \cup Q = Q$, and the rules are those of $\mathcal{P}$, augmented with a rule $\langle q', \varepsilon\rangle \hookrightarrow \langle q, \gamma\rangle$ for each transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}$'s transition set $\rightarrow_0$.

Strictly speaking, this way of combining $\mathcal{P}$ and $\mathcal{A}$ is not a PDS because PDSs do not have rules of the form $\langle q', \varepsilon\rangle \hookrightarrow \langle q, \gamma\rangle$. However, such rules can be accommodated by redefining the transition relation between configurations as follows:

> If $r = \langle q, \gamma\rangle \hookrightarrow \langle q', w\rangle$, then $\langle q, \gamma w'\rangle \xRightarrow{\langle r\rangle} \langle q', ww'\rangle$ for all $w' \in \Gamma^*$.
>
> If $r = \langle q, \varepsilon\rangle \hookrightarrow \langle q', \gamma\rangle$, then $\langle q, w'\rangle \xRightarrow{\langle r\rangle} \langle q', \gamma w'\rangle$ for all $w' \in \Gamma^*$.

Using this extension of PDSs, $\mathcal{A}^R\mathcal{P}$ is defined as follows:

> $\mathcal{A}^R\mathcal{P}$'s states are $P \cup Q \cup \{q_{p',\gamma'} \mid \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta\}$ ($= Q \cup \{q_{p',\gamma'} \mid \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta\}$, because $P \subseteq Q$), and its set of rules Δ' is defined as follows:
>
> (1) For each rule r of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$, Δ' contains r.
> (2) For each rule r of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$, Δ' contains r.
> (3) For each rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$, Δ' contains two rules: $\langle p, \gamma\rangle \hookrightarrow \langle q_{p',\gamma'}, \gamma''\rangle$ and $\langle q_{p',\gamma'}, \varepsilon\rangle \hookrightarrow \langle p', \gamma'\rangle$.
> (4) For each transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}$'s transition set $\rightarrow_0$, Δ' contains a rule $\langle q', \varepsilon\rangle \hookrightarrow \langle q, \gamma\rangle$.

The two rules introduced in item (3) recast a rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ in terms of the extension in which ε is allowed on the left-hand side of a rule. (It is intentional that if rule set Δ has two rules with right-hand sides $\langle p', \gamma'\gamma''\rangle$ and $\langle p', \gamma'\gamma'''\rangle$, only one copy of the rule $\langle q_{p',\gamma'}, \varepsilon\rangle \hookrightarrow \langle p', \gamma'\rangle$ will

Production		for each
(1) $PushSeq_{(q,\gamma,q')}$	$\to \varepsilon$	$(q,\gamma,q') \in \to_0$
(2) $PushSeq_{(p',\gamma',q')}$	$\to PushSeq_{(p,\gamma,q)}\ \ PushSeq_{(q,\gamma',q')}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q, q' \in Q, \gamma' \in \Gamma$
(3) $PushSeq_{(p',\gamma',q)}$	$\to PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
(4) $PushSeq_{(p',\gamma',q_{p',\gamma'})}$	$\to \varepsilon$	$\langle q_{p',\gamma'},\varepsilon\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta'$
(5) $PushSeq_{(q_{p',\gamma'},\gamma'',q)}$	$\to PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle q_{p',\gamma'},\gamma''\rangle \in \Delta', q \in Q$

Fig. 13. A context-free grammar for the push sequences of $\mathcal{A}^R\mathcal{P}$, and the $\mathcal{A}^R\mathcal{P}$ rules that correspond to each production.

appear in Δ'.)

We say that a configuration $c = \langle p, \gamma_1\gamma_2 \ldots \gamma_n\rangle$, for $p \in P$, is *accepted* by $\mathcal{A}^R\mathcal{P}$ if there is a path to c from a configuration $\langle q_f, \varepsilon\rangle$, where $q_f \in F$. Note that because $\mathcal{A}$ has neither ε-transitions nor transitions leading to initial states, $\mathcal{A}^R\mathcal{P}$'s behavior during an accepting run can be divided into two phases: transitions during which $\mathcal{A}^R\mathcal{P}$ mimics $\mathcal{A}$ in reverse—and therefore generates a configuration $c' \in C$, followed by transitions during which $\mathcal{A}^R\mathcal{P}$ mimics $\mathcal{P}$, starting from c'. An accepting run of $\mathcal{A}^R\mathcal{P}$ starts in a state $q_f \in F$; while it remains in states in $(Q \setminus P)$, $\mathcal{A}^R\mathcal{P}$ can only perform a sequence of pushes, possibly reaching a state in P. At the first moment that the run of $\mathcal{A}^R\mathcal{P}$ reaches a configuration c' with a state in P, c' must be a configuration accepted by $\mathcal{A}$ (i.e., c' would be accepted if $\mathcal{A}$ were run in the forward direction), and hence $c' \in C$. During the second phase, $\mathcal{A}^R\mathcal{P}$ mimics transitions of $\mathcal{P}$ to reach configuration c. In other words, $\mathcal{P}$ can reach c starting from configuration c'. Consequently, $\mathcal{A}^R\mathcal{P}$ accepts a configuration c iff $c \in post^*(C)$.

Definition 16 (Push Sequence) A *push sequence* for $q' \in Q$, $\gamma \in \Gamma$, and $q \in Q$ is a sequence of transitions of $\mathcal{A}^R\mathcal{P}$'s transition relation that (i) starts in a configuration $\langle q', w\rangle$, (ii) ends in a configuration $\langle q, \gamma w\rangle$, and (iii) throughout the transition sequence, the stack is always of the form $w'w$ for some (possibly empty) string $w' \in \Gamma^*$, where the contents of w is never inspected during any transition of the transition sequence.

As with pop sequences, there are many similar push sequences that differ only in the untouched part of the stack (i.e., w).

The family of push sequences for a given q', γ, and q can be characterized by the complete derivation trees derived from nonterminal $PushSeq_{(q,\gamma,q')}$, using the grammar shown in Fig. 13. Note that the subscripts and rules in Fig. 13 should be read from right to left: the push sequences for q', γ, and q are characterized by $PushSeq_{(q,\gamma,q')}$. For instance, rule (2) says that if (i) starting in state q' there is a push sequence that ends in q (pushing γ') and (ii) starting in q there is a push sequence that ends in p (pushing γ), then the concatenation of these two sequences, followed by the application of rule $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle$

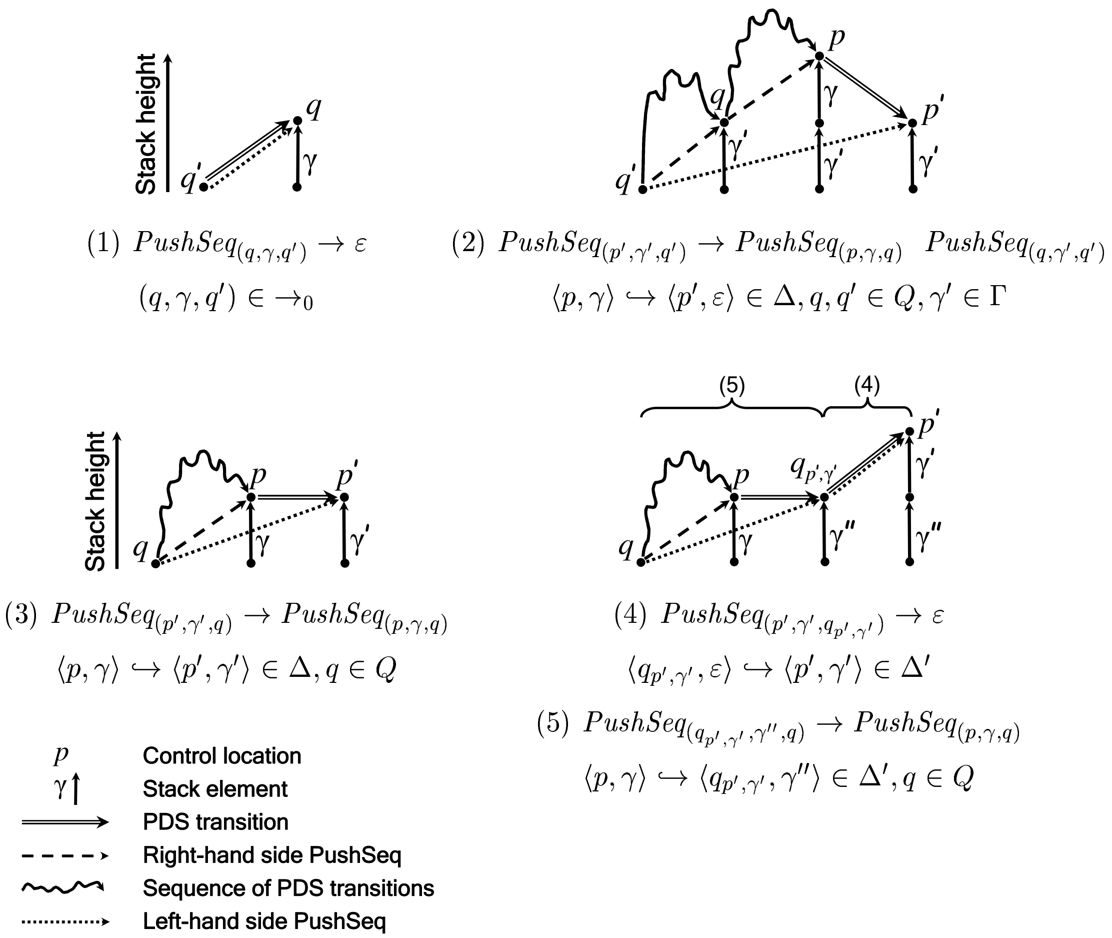


Fig. 14. Schematic diagram of the types of transition sequences captured by the *PushSeq* productions from Fig. 13.

yields a push sequence that starts in state q' and ends in p' (pushing γ'). Fig. 14 depicts the types of transition sequences captured by the *PushSeq* productions from Fig. 13.

Theorem 17 *PDS $\mathcal{A}^R\mathcal{P}$ has a push sequence for q', γ, and q iff nonterminal $PushSeq_{(q,\gamma,q')}$ of the grammar shown in Fig. 13 has a complete derivation tree. Moreover, for each complete derivation tree with root $PushSeq_{(q,\gamma,q')}$, a right-to-left postorder listing of the derivation tree's production instances (where Fig. 13 defines the correspondence between productions and PDS rules) gives a sequence of rules for a push sequence for q', γ, and q; and every such sequence of rules has a derivation tree with root $PushSeq_{(q,\gamma,q')}$.*

Proof [Sketch] The argument is by induction on push-sequence length. Each of the grammar rules of Fig. 13, when the right-hand side is read right to left, followed by an application of the corresponding PDS rule shown in the last column of Fig. 13, results in a push sequence corresponding to the left-hand-side nonterminal symbol (see Fig. 14). Moreover, each push sequence of $\mathcal{A}^R\mathcal{P}$

	Production		for each
(1)	$PushSeq_{(q,\gamma,q')}$	$\to \epsilon$	$(q,\gamma,q') \in \to_0$
(2)	$SameLevelSeq_{(p',\varepsilon,q)}$	$\to PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in Q$
(2′)	$PushSeq_{(p',\gamma',q')}$	$\to SameLevelSeq_{(p',\varepsilon,q)}\ \ PushSeq_{(q,\gamma',q')}$	$p' \in P,\ q,q' \in Q$
(3)	$PushSeq_{(p',\gamma',q)}$	$\to PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
(4)	$PushSeq_{(p',\gamma',q_{p',\gamma'})}$	$\to \epsilon$	$\langle q_{p',\gamma'},\varepsilon\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta'$
(5)	$PushSeq_{(q_{p',\gamma'},\gamma'',q)}$	$\to PushSeq_{(p,\gamma,q)}$	$\langle p,\gamma\rangle \hookrightarrow \langle q_{p',\gamma'},\gamma''\rangle \in \Delta', q \in Q$

Fig. 15. A refactoring of the grammar from Fig. 13. In particular, rules (2) and (2′) above correspond to rule (2) of Fig. 13.

must end with an application of a rule of $\mathcal{A}^R\mathcal{P}$, and hence can be decomposed according to rules (1)–(5) of Fig. 13. □

In what follows, we will work with the *PushSeq* grammar shown in Fig. 15, rather than than the one shown in Fig. 13. In Fig. 15, the only change is that a new family of nonterminals is introduced, denoted by $SameLevelSeq_{(p',\varepsilon,q)}$, and production (2) from Fig. 13 is broken into two productions: (2) and (2′). (This refactoring is introduced so that the $post^*$ algorithm that we finally end up with—Alg. 3—closely resembles the $post^*$ algorithm from Schwoon's thesis [11, Alg. 2].)

To capture the set $PostStarWitnesses(\langle p,\gamma_1\gamma_2\ldots\gamma_n\rangle, C)$, where C is recognized by automaton $\mathcal{A}$, we extend the context-free grammar from Fig. 15 by the set of productions

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

This language captures all ways in which PDS $\mathcal{A}^R\mathcal{P}$ can accept $\langle p,\gamma_1\gamma_2\ldots\gamma_n\rangle$: the set of $post^*$ 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 $PushSeq_{(q_{i-1},\gamma_i,q_i)}$ gives the push sequence that $\mathcal{A}^R\mathcal{P}$ performs to generate symbol γ_i. (If there are no $post^*$ witnesses for $\langle p,\gamma_1\gamma_2\ldots\gamma_n\rangle$, there are no complete derivation trees with root $Accepted[\gamma_1\gamma_2\ldots\gamma_n]_{(p)}$.)

We are concerned with weighted pushdown systems, and thus consider the weighted version of $\mathcal{A}^R\mathcal{P}$, denoted by $\mathcal{A}^R\mathcal{W}$, in which the rules in Δ' receive weights as follows:

(1) For each rule r of the form $\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta$, Δ' contains r with weight $f(r)$.

(2) For each rule r of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$, Δ' contains r with weight $f(r)$.
(3) For each rule r of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$, Δ' contains two rules: $\langle p, \gamma \rangle \hookrightarrow \langle q_{p',\gamma'}, \gamma'' \rangle$ with weight $f(r)$ and $\langle q_{p',\gamma'}, \varepsilon \rangle \hookrightarrow \langle p', \gamma' \rangle$ with weight 1.
(4) For each transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}$'s transition set $\rightarrow_0$, Δ' contains a rule $\langle q', \varepsilon \rangle \hookrightarrow \langle q, \gamma \rangle$ with weight 1.

As argued earlier, $\mathcal{A}^R\mathcal{W}$ accepts a configuration c iff $c \in post^*(C)$. Because the complete derivation trees with root $Accepted[\gamma_1\gamma_2 \ldots \gamma_n]_{(p)}$ encode the transition sequences by which $\mathcal{A}^R\mathcal{W}$ accepts $\langle p, \gamma_1\gamma_2 \ldots \gamma_n \rangle$, to cast a GPS problem as a grammar problem, we have to attach appropriate production functions to the productions of Fig. 15 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 Fig. 16. Notice that functions $h_{2'}$ and h_6 reverse the order of their arguments, and h_2, h_3, and h_5 place $f(r)$ at the right-hand end of the semiring-product expression. This corresponds to the fact that for $\mathcal{A}^R\mathcal{W}$ to accept $\langle p, \gamma_1\gamma_2 \ldots \gamma_n \rangle$, it must perform push sequences in the order γ_n, ..., γ_1: each grammar rule's left-hand-side push sequence requires that the push sequences of the right-hand side be performed right to left, followed by an application of the corresponding WPDS rule (cf. Theorem 17).

As in Section 3.1, not all instantiations of the productions listed in Fig. 15 and Eqn. (1) are relevant to the final solution; we want to prevent the algorithm from exploring useless nonterminals of the grammar from Fig. 15 and Eqn. (1). Exploring the *PushSeq* subgrammar lazily saves us from having to construct the entire *PushSeq* subgrammar.

Moreover, all path questions with respect to a given source-configuration set C involve the same subgrammar for the *PushSeq* nonterminals. Consequently, the information about whether a complete derivation tree with root nonterminal $PushSeq_{(q,\gamma,q')}$ exists (i.e., whether $PushSeq_{(q,\gamma,q')}$ is a productive nonterminal) can be precomputed and returned in the form of an automaton. Productive nonterminals represent automaton transitions, and the productions that involve any given transition can be constructed on-the-fly, as is done in Alg. 3, shown in Fig. 17.

Alg. 3 finds the productive *PushSeq* and *SameLevelSeq* nonterminals in the grammar from Fig. 15:[7] *workset* contains the set of transitions (nonterminals) still to be considered, and the algorithm iterates until *workset* is empty.

[7] There is one slight exception to this statement. Nonterminals of the form $PushSeq_{((p',\gamma',q_{p',\gamma'}))}$ can only derive ε, and hence are always productive. However, line 20 treats such nonterminals lazily; they are only placed in the transition set if there is a productive nonterminal of the form $PushSeq_{((q_{p',\gamma'},\gamma'',q))}$ (see line 21).

	Production		for each
(1)	$PushSeq_{(q,\gamma,q')}$	$\to h_1(\epsilon)$	$(q,\gamma,q') \in \to_0$
	$h_1 = 1$		
(2)	$SameLevelSeq_{(p',\varepsilon,q)}$	$\to h_2(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle \in \Delta, q \in Q$
	$h_2 = \lambda x.x \otimes f(r)$		
(2′)	$PushSeq_{(p',\gamma',q')}$	$\to h_{2'}(SameLevelSeq_{(p',\varepsilon,q)}\ \ PushSeq_{(q,\gamma',q')})$	
			$p' \in P,\ q,q' \in Q$
	$h_{2'} = \lambda x.\lambda y.y \otimes x$		
(3)	$PushSeq_{(p',\gamma',q)}$	$\to h_3(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \in \Delta, q \in Q$
	$h_3 = \lambda x.x \otimes f(r)$		
(4)	$PushSeq_{(p',\gamma',q_{p',\gamma'})}$	$\to h_4(\epsilon)$	$\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta$
	$h_4 = 1$		
(5)	$PushSeq_{(q_{p',\gamma'},\gamma'',q)}$	$\to h_5(PushSeq_{(p,\gamma,q)})$	$r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \in \Delta, q \in Q$
	$h_5 = \lambda x.x \otimes f(r)$		
(6)	$Accepting[\gamma_1\gamma_2\ldots\gamma_n]_{(p,q)}$	$\to h_6(PushSeq_{(p,\gamma_1,q_1)}, PushSeq_{(q_1,\gamma_2,q_2)}, \ldots, PushSeq_{(q_{n-1},\gamma_n,q)})$	
			$p \in P, q_i \in Q,$ for $1 \le i \le n-1,$ and $q \in F$
	$h_6 = \lambda x_1.\lambda x_2 \ldots \lambda x_n.x_n \otimes \ldots \otimes x_2 \otimes x_1$		
(7)	$Accepted[\gamma_1\gamma_2\ldots\gamma_n]_{(p)}$	$\to h_7(Accepting[\gamma_1\gamma_2\ldots\gamma_n]_{(p,q)})$	
			$p \in P, q \in F$
	$h_7 = \lambda x.x$		

Fig. 16. An abstract grammar problem for the weighted *post** problem.

Lines 11, 17, 18, 20, and 21 process the productions of types (1), (2), (3), (4), and (5), respectively. Lines 23 and 25 handle the productions of type (2′).

As in Section 3.1.3, the time complexity increases from the $\mathcal{O}(n_P n_\Delta(n_1+n_2)+n_P n_0)$ complexity of the unweighted case [11] (where $n_P = |P|$, $n_\Delta = |\Delta|$, $n_Q = |Q|$, $n_0 = |{\to_0}|$, $n_1 = |Q\backslash P|$, and n_2 is the number of different pairs (p',γ') such that there is a rule of the form $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle$ in Δ) by a factor that is no more than the length of the maximal-length descending chain to any value that appears in the annotated automaton.

3.2.1 Generation of Witness Sets

By analogy with Section 3.1.4, which provides a method for obtaining witness sets for GPP problems, this section discusses how to extend Alg. 3 to allow the recovery of witness sets for GPS problems.

The basic idea behind this extension is to adapt the method from Section 3.1.4,

Algorithm 3

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 $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ that accepts C, such that
$\mathcal{A}$ has no transitions into P states and has no ε-transitions.

Output: a $\mathcal{P}$-automaton $\mathcal{A}_{post^*} = (Q', \Gamma, \rightarrow, P, F)$ with ε-transitions
that accepts $post^*(C)$;
a function l that maps every $(q, \gamma, q') \in \rightarrow$ to the value of
$m_G(PushSeq_{(q,\gamma,q')})$ in the abstract grammar problem defined in Fig. 16.

procedure $update(t, v)$
begin
    $\rightarrow := \rightarrow \cup \{t\}$
    $newValue := l(t) \oplus v$
    $changed := (newValue \neq l(t))$
    if $changed$ **then**
        $workset := workset \cup \{t\}$
        $l(t) := newValue$
end

$\rightarrow := \rightarrow_0$; $workset := \rightarrow_0$; $l := \lambda t.0$
for all $t \in \rightarrow_0$ **do** $l(t) := 1$
$Q' := Q$; **for all** $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$ **do** $Q' := Q' \cup \{q_{p',\gamma'}\}$
while $workset \neq \emptyset$ **do**
    select and remove a transition $t = (p, \gamma, q)$ from $workset$
    if $\gamma \neq \varepsilon$ **then**
        for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $update((p', \varepsilon, q), l(t) \otimes f(r))$
        for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ **do** $update((p', \gamma', q), l(t) \otimes f(r))$
        for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$ **do**
            $update((p', \gamma', q_{p',\gamma'}), 1)$
            $update((q_{p',\gamma'}, \gamma'', q), l(t) \otimes f(r))$
            if $changed$ **then**
                for all $t' = (p'', \varepsilon, q_{p',\gamma'})$ **do** $update((p'', \gamma'', q), l(t) \otimes f(r) \otimes l(t'))$
    else
        for all $t' = (q, \gamma', q') \in \rightarrow$ **do** $update((p, \gamma', q'), l(t') \otimes l(t))$
return $((Q', \Gamma, \rightarrow, P, F), l)$

Fig. 17. An algorithm for creating a weighted automaton for the weighted $post^*$ problem.

which records pop sequences, to record push sequences: we create a hypergraph $\mathcal{G}_{push} = (N, E)$, where $N = (\rightarrow \times D)$ and $E = (N \times \Delta' \times N^*)$. Note that for GPS problems we use $\Delta' = \Delta \cup \{\varepsilon\}$; i.e., we allow hyperedges to be labeled with either a rule or ε, where the latter can be read as "no rule".

A node $n = (t, d) \in N$, where $t = (p, \gamma, q)$, records that there exist push sequences $\sigma_1, \ldots, \sigma_k$ for q, γ, p and $d = \bigoplus_{i=1}^{k} d_i$, where $d_1, \ldots, d_k$ are the semiring values accumulated along these paths. If $t \in \rightarrow_0$ is a transition from $\mathcal{A}$ and

$d = 1$, then $k = 1$ and $\sigma_1 = \varepsilon$; this fact is represented by a node $(t, 1)$ that has no incoming hyperedges. For a node $n = (t, d)$ for which $t \notin \rightarrow_0$ is not a transition from $\mathcal{A}$, each hyperedge $(n, r', n_1 \dots n_m)$ corresponds to a collection of push sequences for q, γ, p; each of these push sequences is of the form $\tau_m \dots \tau_1 r'$, where each τ_i, for $1 \leq i \leq m$, is a push sequence for n_i. [8]

Once $\mathcal{G}_{push}$ is constructed, the information in it captures a witness set for any given configuration c: if $t_1 \dots t_m$ is a path in $\mathcal{A}_{post^*}$ by which c is accepted, then $\omega(c)$ consists of every sequence $\sigma_m \dots \sigma_1$, where, for every $1 \leq i \leq m$, σ_i is a push sequence for $(t_i, l(t_i))$.

The necessary changes to Alg. 3 are as follows: In line 11, the empty hypergraph is created by setting $N := \emptyset$ and $E := \emptyset$. In line 12, a node $(t, 1)$ is added to N for every $t \in \rightarrow_0$. For the *update* procedure, we re-use Alg. 2 from Section 3.1.4 with two small modifications:

- Because Alg. 3 uses the variable *changed*, line 5 is replaced by

$$\begin{array}{l} \mathit{changed} := (\mathit{newValue} \neq l(t)) \\ \textbf{if } \neg \mathit{changed} \textbf{ then return} \end{array}$$

- In line 12, the $\otimes$ operator is applied in reverse order:

$$\textbf{if } v \not\sqsubseteq \left(\bigotimes_{i=1}^{m} d_{m-i+1} \right) \otimes f(r') \textbf{ then } \dots$$

  (where $f(\varepsilon) = 1$).

Compared to Alg. 3, the new *update* procedure takes two additional arguments: r, the rule, and T, the list of transitions used for the addition. The calls on *update* in Fig. 17 are modified as follows:

line 17:	$\mathit{update}((p', \varepsilon, q), l(t) \otimes f(r), r, t)$
line 18:	$\mathit{update}((p', \gamma', q), l(t) \otimes f(r), r, t)$
line 20:	$\mathit{update}((p', \gamma', q_{p',\gamma'}), 1, \varepsilon, \varepsilon)$
line 21:	$\mathit{update}((q_{p',\gamma'}, \gamma'', q), l(t) \otimes f(r), r, t)$
line 23:	$\mathit{update}((p'', \gamma'', q), l(t) \otimes f(r) \otimes l(t'), \varepsilon, t'\, t)$
line 25:	$\mathit{update}((p, \gamma', q'), l(t') \otimes l(t), \varepsilon, t\, t')$

[8] When comparing this definition with that from Section 3.1.4, note that in addition to substituting push sequences for pop sequences, we also change the order in which push sequences are assembled from the source nodes of the hyperedges.

The choice of the arguments for lines 17, 18, and 21 is self-explanatory. In lines 23 and 25, we create a new transition by "contracting" two transitions, i.e., without firing a pushdown rule; hence, we use ε for the rule. The only complicated case is line 20. Here, recall that the rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ is responsible for one production of type (4), dealt with in line 20, and another of type (5), processed in line 21. Because line 21 is already recording the fact that r was applied, both arguments in line 20 are ε.

4 Applications to Interprocedural Dataflow Analysis

This section describes the application of weighted PDSs to interprocedural dataflow analysis. The approach presented here has been used to create analyses for a variety of problems, including uninitialized variables, live variables, linear constant propagation [6], and affine-relation analysis [7].

This section (as well as Section 5) illustrates several classic concepts in interprocedural dataflow analysis from the vantage point of weighted PDSs. It also illustrates how algorithms from Section 3 provide a way to generalize previous frameworks for interprocedural dataflow analysis [19,6].

The presentation is divided into four parts. Section 4.1 presents background material on interprocedural dataflow analysis. Section 4.2 discusses how conventional dataflow information can be obtained by formulating dataflow-analysis problems as GPR problems. Section 4.3 shows how information that goes beyond what conventional dataflow-analysis algorithms provide can be obtained by solving GPR problems. Section 4.4 discusses extensions.

4.1 Background on Interprocedural Dataflow Analysis

Dataflow analysis is concerned with determining an appropriate dataflow value to associate with each node n in a program, to summarize (safely) some aspect of the possible memory configurations that hold whenever control reaches n. To define an instance of a dataflow problem, one needs

- The control-flow graph for the program.
- A meet semilattice $(V, \sqcap)$ with greatest element $\top$:
  - · Elements of V represent sets of possible memory configurations. Each point in the program is to be associated with some member of V.
  - · The meet operator $\sqcap$ is used for combining information obtained along different paths.

- A value $v_0 \in V$ that represents the set of possible memory configurations at the beginning of the program.
- An assignment of dataflow transfer functions (of type $V \to V$) to the edges of the control-flow graph.

When $(D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, $(D, \oplus)$ is a meet semilattice. However, when interprocedural dataflow-analysis problems are formulated as GPR problems, D is $V \to V$, not V. Consequently, we will use $(V, \sqcap)$ and $\top$ when we wish to emphasize that we are discussing dataflow *values*, and $(D, \oplus, \otimes, 0, 1)$ when we turn to GPR encodings of dataflow-analysis problems.

Typically, a dataflow-analysis problem is formulated as a *path-function problem*: the path function pf_q for path q is the composition of the transfer functions that label the edges of q. In *intra*procedural dataflow analysis, the goal is to determine, for each node n, the "meet-over-*all-paths*" solution:

$$\mathrm{MOP}_n = \bigsqcap_{q \in \mathrm{Paths}(\mathrm{enter}, n)} \mathrm{pf}_q(v_0),$$

where Paths(enter, n) denotes the set of paths in the control-flow graph from the enter node to n [20].[9] MOP_n represents a summary of the possible memory configurations that can arise at n: because $v_0 \in V$ represents the set of possible memory configurations at the beginning of the program, $\mathrm{pf}_q(v_0)$ represents the contribution of path q to the memory configurations summarized at n.

*Inter*procedural dataflow-analysis problems are often defined in terms of a program's *supergraph* (or "interprocedural control-flow graph"), an example of which is shown in Fig. 18. A supergraph consists of a 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,[10] 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) and (n_6, n_7) in Fig. 18). 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 interprocedural *call-to-enter* edge from c to the enter node of the called procedure; an interprocedural *exit-to-return-site* edge from

[9] For some dataflow-analysis problems, such as constant propagation, the meet-over-all-paths solution is uncomputable. A sufficient condition for the solution to be computable is for each transfer function f to distribute over the meet operator; that is, for all $a, b \in V$, $f(a \sqcap b) = f(a) \sqcap f(b)$.

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

```
int g, h;

void main() {
 n1: g = h = 0;
 n2,n3: p();
 n4: if(...) {
  n5: g = 1;
  n6,n7: p();
 }
 n8: ...;
 return;
}

void f() {
 n9: ...;
 return;
}
```

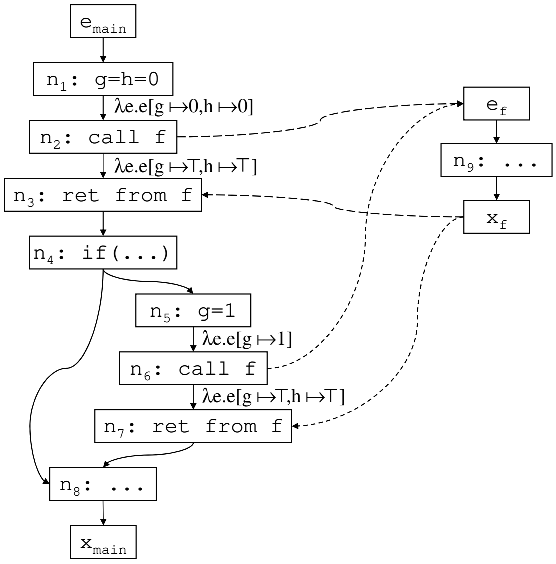


Fig. 18. A program fragment and its supergraph. The environment transformer for all unlabeled edges is $\lambda e.e$.

the exit node of the called procedure to r; an intraprocedural *call-to-return-site* edge from c to r.[11]

Definition 18 *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 19 *The sets of* **same-level valid paths** *and* **valid paths** *in a supergraph are defined inductively as follows:*

[11] The call-to-return-site edges are included so that programs with local variables and parameters can be handled. Functions on call-to-return-site edges extract (from the dataflow information valid immediately before the call) dataflow information about local variables that must be re-established after the return from the call. The dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables that holds at the call site to be combined with the information about global variables that holds at the end of the called procedure.

- *The empty path is a* **same-level valid path** *(and therefore a* **valid path***).*
- *Path $p \parallel [e]$ 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 $p = p_h \parallel [e_c] \parallel p_t$, 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 4.1 *In the supergraph shown in Fig. 18, the path*

$$e_{main} \rightarrow n_1 \rightarrow n_2 \rightarrow e_f \rightarrow n_9 \rightarrow x_f \rightarrow n_3 \rightarrow n_4$$

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

$$e_{main} \rightarrow n_1 \rightarrow n_2 \rightarrow e_f \rightarrow n_9$$

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

$$e_{main} \rightarrow n_1 \rightarrow n_2 \rightarrow e_f \rightarrow n_9 \rightarrow x_f \rightarrow n_7$$

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

In interprocedural dataflow analysis, the goal shifts from finding the meet-over-*all-paths* solution to the more precise "meet-over-*all-valid-paths*", or "context-sensitive" solution. 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 analysis captures the fact that the results propagated back to each return-site r should depend on the memory configurations that arise at the call site that corresponds to r, but not on the memory configurations that arise at call sites that do not correspond to r. More precisely, the goal of a context-sensitive analysis is to find the *meet-over-all-valid-paths* value for nodes of the supergraph [19,21,6]:

$$\mathrm{MOVP}_n = \prod_{q \in \mathrm{VPaths}(\mathrm{enter}_{\mathrm{main}}, n)} \mathrm{pf}_q(v_0),$$

where $\mathrm{VPaths}(\mathrm{enter}_{\mathrm{main}}, n)$ denotes the set of valid paths from the main procedure's enter node to n.

Although some valid paths may also be infeasible execution paths, none of the non-valid paths are feasible execution paths. By restricting attention to just the valid paths from $\mathrm{enter}_{\mathrm{main}}$, we thereby exclude some of the infeasible execution paths. In general, therefore, MOVP_n characterizes the memory configurations at n more precisely than MOP_n.

Local Variables

Call-to-return-site edges introduce some additional paths in the supergraph that do not correspond to standard program-execution paths. The intuition behind this approach is that the interprocedurally valid paths of Defn. 19 correspond to "paths of action" for particular *subsets* of the runtime entities (e.g., global variables). The path function along a particular path contributes only *part* of the dataflow information that reflects what happens during the corresponding run-time execution. The facts for other subsets of the runtime entities (e.g., local variables) are handled by different "trajectories", for example, paths that take "short-cuts" via call-to-return-site edges.

The use of call-to-return-site edges is less precise that some other approaches to handling local variables, such as the method proposed by Knoop and Steffen [21] and the method used in Moped [22], both of which have a bit of the flavor of a relational analysis. (See Lal et al. [23] for a description of how to generalize WPDSs so that local variables can be handled in the more precise, relational manner.)

The Examples Used in Later Sections

In the remainder of the paper, we illustrate the application of weighted PDSs to interprocedural dataflow analysis using instances of two dataflow-analysis problems: simple constant propagation and linear constant propagation. Our choices were motivated by wanting to use the simplest example possible to illustrate the benefit of the various methods presented in the paper.

- Section 4.2 illustrates how conventional dataflow information can be obtained by solving GPR problems, using simple constant propagation.
- Section 4.3 uses linear constant propagation to illustrate how information that goes beyond what conventional dataflow-analysis algorithms provide can be obtained by solving GPR problems.
- Section 5 returns to simple constant propagation to illustrate differential algorithms for solving GPR problems.

It should also be noted that even though these problems are all examples of independent-attribute problems, the weighted PDS approach also applies to relational analyses. (For material on independent-attribute and relational analyses, see [24, Chapter 4].)

This section shows how conventional dataflow information can be obtained by formulating dataflow-analysis problems as generalized pushdown reachability problems. This method applies to distributive dataflow-analysis problems for which the transfer functions are "composable" and are drawn from a meet semilattice that has no infinite descending chains; that is,

- There must be finite representations for all elements in the set of functions F that consists of the basic dataflow-transfer functions, closed under meet and composition.
- F must form a meet semilattice with no infinite descending chains.
- The functions in F must be strict in the value $\top$. This ensures that the function $\lambda x.\top$ is an annihilator for $\odot$, where $\odot$ is the reversal of function composition—i.e., $f \odot g = g \circ f$, where $(g \circ f)(x) = g(f(x))$. Note that one consequence of this is that transfer functions that would otherwise be constant (e.g., $\lambda x.c$) must be modified to return $\top$ if the argument is $\top$ and c otherwise. [12]

For such problems, the semiring that will be used is $(F, \sqcap, \odot, \lambda x.\top, \lambda x.x)$.

The supergraph of each program to be analyzed is encoded as a weighted PDS with the following properties:

- There is a single control location p
- Each supergraph node n is represented by a separate stack symbol γ_n.
- Each intraprocedural edge, call site, and return statement is represented by a WPDS rule:

$\langle p, \gamma_n \rangle \hookrightarrow \langle p, \gamma_{n'} \rangle$	intraprocedural edge from n to n'
$\langle p, \gamma_n \rangle \hookrightarrow \langle p, \gamma_{n'}\gamma_{n''} \rangle$	procedure call from call site n to n', with return-site n''
$\langle p, \gamma_n \rangle \hookrightarrow \langle p, \varepsilon \rangle$	return statement n

Each rule is weighted by the semiring element for the appropriate dataflow transfer function. It is not hard to see that with this encoding each *valid path* in the program's supergraph corresponds to a *path* in the PDS's transition system, and vice versa.

By applying Alg. 3 to an automaton that represents the single configuration $\langle p, \mathrm{enter}_{\mathrm{main}} \rangle$, which corresponds to the initial configuration of the program (i.e., with no stacked return nodes), we can create an automaton that contains

[12] An alternative approach is to drop the requirement that the functions in F be strict in $\top$, but define $\odot$ in terms of a variant of $\circ$ that is strict in $\lambda x.\top$.

Algorithm 4
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 $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ that accepts C, such that
$\mathcal{A}$ has no transitions into P states and has no ε-transitions.

Output: for each $p \in P$ and $\gamma \in \Gamma$, compute $V_{p,\gamma} := \bigoplus\{\, \delta(\langle p, \gamma w\rangle) \mid w \in \Gamma^* \,\}$

Let $\mathcal{A}_{post^*} = (Q', \Gamma, \rightarrow, P, F)$ and l be the output from calling Alg. 3

for all $q \in Q' \setminus (P \cup F)$ **do** $W_q := 0$
for all $q \in F$ **do** $W_q := 1$
workset $:= F$
while *workset* $\neq \emptyset$ **do**
  select and remove a state q from *workset*
  oldValue $:= W_q$
  $W_q := \bigoplus_{t=(q,\gamma,q') \in \rightarrow} (W_{q'} \otimes l(t))$
  if *oldValue* $\neq W_q$ **then** *workset* $:=$ *workset* $\cup \{\, q' \mid q' \notin P,\ (q', \gamma, q) \in \rightarrow \}$

for all $p \in P$ and $\gamma \in \Gamma$ **do** $V_{p,\gamma} := \bigoplus_{t=(p,\gamma,q) \in \rightarrow} (W_q \otimes l(t))$

Fig. 19. An algorithm that, for each $p \in P$ and $\gamma \in \Gamma$, computes $V_{p,\gamma}$.

information about $\delta(c)$ for all reachable configurations c. From this automaton, we wish to obtain the conventional meet-over-all-valid-paths value for each supergraph node n. This could be done by performing repeated queries on the automaton—one query for each node n. However, the answers can be obtained more efficiently (and in a more general setting) by performing a fixed-point computation on the automaton itself, as described below.

Given a weighted pushdown system $\mathcal{W}$ and an automaton $\mathcal{A}$ that accepts configurations in C, Alg. 4 computes, for each $p \in P$ and each $\gamma \in \Gamma$, the value $V_{p,\gamma} = \bigoplus\{\, \delta(\langle p, \gamma w\rangle) \mid w \in \Gamma^* \,\}$, where δ is defined as for the generalized pushdown successor problem in Defn. 7. That is,

$$\begin{aligned} V_{p,\gamma} &= \bigoplus\{\, \delta(\langle p, \gamma w\rangle) \mid \langle p, \gamma w\rangle \in post^*(C) \,\} \\ &= \bigoplus\{\, v(\sigma) \mid \sigma \in path(c, \langle p, \gamma w\rangle), c \in C, w \in \Gamma^* \,\} \end{aligned} \tag{2}$$

and $V_{p,\gamma} = 0$ if there is no $w \in \Gamma^*$ such that $\langle p, \gamma w\rangle \in post^*(C)$.

We now sketch a proof that $V_{p,\gamma_n}(v_0) = \mathrm{MOVP}_n$, where v_0 represents the set of possible memory configurations at $\mathrm{enter}_{\mathrm{main}}$.

Theorem 20 *Let $\mathcal{W}$ be a weighted pushdown system that encodes a dataflow problem on supergraph G, as described earlier in this section. Let $\mathcal{A}$ be an automaton that accepts the single configuration $\langle p, enter_{main}\rangle$. Then, for the output of Alg. 4, for each supergraph node n, we have*

$$V_{p,\gamma_n}(v_0) = MOVP_n,$$

where v_0 represents the set of possible memory configurations at $enter_{main}$.

Proof [Sketch] For every valid path t in G from $\text{enter}_{\text{main}}$ to vertex n, there is a sequence σ of rules by which $\mathcal{W}$, starting in configuration $\langle p, \text{enter}_{\text{main}} \rangle$, reaches a configuration $\langle p, \gamma_n w \rangle$, and vice versa. Furthermore, $\text{pf}_t = v(\sigma)$.

The proof has two parts: first, we show that $\text{MOVP}_n \sqsubseteq V_{p,\gamma_n}(v_0)$. Let S be a finite set of paths in G such that

$$\bigsqcap \{\, \text{pf}_s(v_0) \mid s \in S \,\} = \text{MOVP}_n$$

S must exist because we have restricted ourselves to WPDS problems in which semirings have no infinite descending chains. Let $T = \{\, \sigma \mid \sigma \text{ corresponds to } s, s \in S \,\}$. For every $\sigma \in T$, σ drives the weighted PDS from $\langle p, \text{enter}_{\text{main}} \rangle$ to $\langle p, \gamma_n w \rangle$, for some $w \in \Gamma^*$. Then we have

$$\begin{aligned}
\text{MOVP}_n &= \bigsqcap \{\, \text{pf}_s(v_0) \mid s \in S \,\} \\
&= \left(\bigsqcap \{\, \text{pf}_s \mid s \in S \,\} \right)(v_0) \\
&= \left(\bigoplus \{\, v(\sigma) \mid \sigma \in T \,\} \right)(v_0) \\
&\sqsubseteq V_{p,\gamma_n}(v_0) && \text{by Eqn. (2)}
\end{aligned}$$

The second part of the proof uses a similar argument in the reverse direction to show that $V_{p,\gamma_n}(v_0) \sqsubseteq \text{MOVP}_n$. It follows that $V_{p,\gamma_n}(v_0) = \text{MOVP}_n$. $\square$

Alg. 4 has three phases:

- In phase 1 (see line 1), it computes $\mathcal{A}_{post^*}$.
- In phase 2 (lines 3–10), for every non-initial state q, the value W_q is computed. Conceptually, W_q is obtained by following all accepting paths starting at state q, accumulating a semiring value for each path, and then taking the meet over all values. In the implementation, this is achieved by a fixed-point computation that propagates values backwards over the transitions of $\mathcal{A}_{post^*}$. This phase does not need to consider the ε-transitions in $\mathcal{A}_{post^*}$ because the weights on all ε-transitions have already been propagated to non-ε-transitions during Alg. 3.
- In phase 3 (line 12), the algorithm computes $V_{p,\gamma}$ for each $p \in P$ and $\gamma \in \Gamma$.

For other applications of weighted PDSs, a similar algorithm could be given that works with *pre**—the main difference would be that the order of the operands of $\otimes$ in lines 9 and 12 would have to be swapped.

Example 4.2 *Fig. 18 shows an instance of constant propagation, a classic dataflow-analysis problem. For constant propagation, a dataflow value con-*

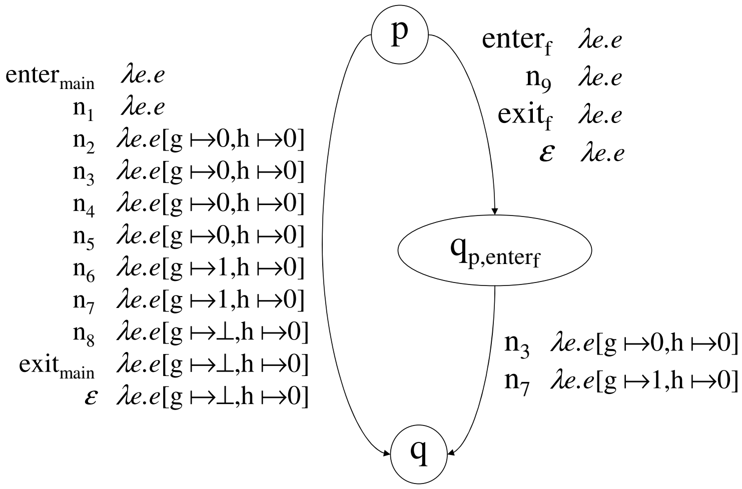


Fig. 20. The automaton $\mathcal{A}_{post^*}$ computed by Alg. 3 for input automaton $\mathcal{A} = (\{p, q\}, \Gamma, \{(p, \text{entry}_{\text{main}}, q)\}, \{p\}, \{q\})$ and the weighted PDS that corresponds to Fig. 18. Each edge represents multiple transitions, one for each line of the label on the edge.

sists of an environment transformer. For this problem, an environment maps each program variable to a value in $\mathbb{Z}_\perp^\top$, the integers extended with $\perp$ (signifying a non-constant value) and $\top$ (signifying no value, or "unmapped"). An environment transformer takes an environment as input and outputs an environment where some variables may be mapped to a new value, and others are mapped to their previous value. A variable that is mapped to $\top$ cannot be updated with a different value. Thus, we have

$$(\lambda e.e[g \mapsto 3, h \mapsto 5])([g \mapsto \top, h \mapsto 2]) = [g \mapsto \top, h \mapsto 5]$$

Here, the environment transformer succeeds in updating the environment for h, but not for g.

Environment transformers are assigned to program statements according to the following table:

statement	*transformer*
`x = k;`	$\lambda e.e[x \mapsto k]$
`x = ...; /* non-constant expression */`	$\lambda e.e[x \mapsto \perp]$
other	$\lambda e.e$

The semiring used for the constant-propagation problem is defined as follows: 0 is $\lambda e.\lambda v.\top$; 1 is the identity function, $\lambda e.e$; the operations $\oplus$ and $\otimes$ are *defined by*

$$f_1 \oplus f_2 = \lambda e.f_1(e) \sqcap f_2(e)$$
$$f_1 \otimes f_2 = f_2 \circ f_1$$

where $e_1 \sqcap e_2$ *on environments* e_1, e_2 *is defined component-wise using the* $\sqcap$ *operator for* $\mathbb{Z}_\perp^\top$*:*

$$u \sqcap v = \begin{cases} u & \text{if } u = v \text{ or } v = \top \\ v & \text{if } u = \top \\ \perp & \text{if } u \neq v \text{ and } u \neq \top \text{ and } v \neq \top \end{cases}$$

The rules for the weighted PDS that encodes the constant-propagation problem for the program shown in Fig. 18 are as follows:

$$\begin{array}{llll}
\langle p, e_{main}\rangle \hookrightarrow \langle p, n_1\rangle & \lambda e.e & \langle p, n_6\rangle \hookrightarrow \langle p, e_f\, n_7\rangle & \lambda e.e \\
\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle & \lambda e.e[g \mapsto 0, h \mapsto 0] & \langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle & \lambda e.e[g \mapsto \top, h \mapsto \top] \\
\langle p, n_2\rangle \hookrightarrow \langle p, e_f\, n_3\rangle & \lambda e.e & \langle p, n_7\rangle \hookrightarrow \langle p, n_8\rangle & \lambda e.e \\
\langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle & \lambda e.e[g \mapsto \top, h \mapsto \top] & \langle p, n_8\rangle \hookrightarrow \langle p, x_{main}\rangle & \lambda e.e \\
\langle p, n_3\rangle \hookrightarrow \langle p, n_4\rangle & \lambda e.e & \langle p, x_{main}\rangle \hookrightarrow \langle p, \varepsilon\rangle & \lambda e.e \\
\langle p, n_4\rangle \hookrightarrow \langle p, n_5\rangle & \lambda e.e & \langle p, e_f\rangle \hookrightarrow \langle p, n_8\rangle & \lambda e.e \\
\langle p, n_4\rangle \hookrightarrow \langle p, x_{main}\rangle & \lambda e.e & \langle p, n_9\rangle \hookrightarrow \langle p, x_f\rangle & \lambda e.e \\
\langle p, n_5\rangle \hookrightarrow \langle p, n_6\rangle & \lambda e.e[g \mapsto 1] & \langle p, x_f\rangle \hookrightarrow \langle p, \varepsilon\rangle & \lambda e.e
\end{array}$$

The input automaton $\mathcal{A}$ *that accepts* $\langle p, enter_{main}\rangle$ *together with the output automaton* $\mathcal{A}_{post^*}$ *obtained for this weighted PDS using post* are given in Fig. 20. From* $\mathcal{A}_{post^*}$*, it is straightforward to compute the results of Alg. 4. For example,*

$$V_{p,n_8} = \lambda e.e[g \mapsto \perp][h \mapsto 0].$$

Thus, at n_8*,* g *has a non-constant value, and* h *has the value* 0*.*

4.3 Obtaining More Than Conventional Dataflow Information

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

```
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;
}
```

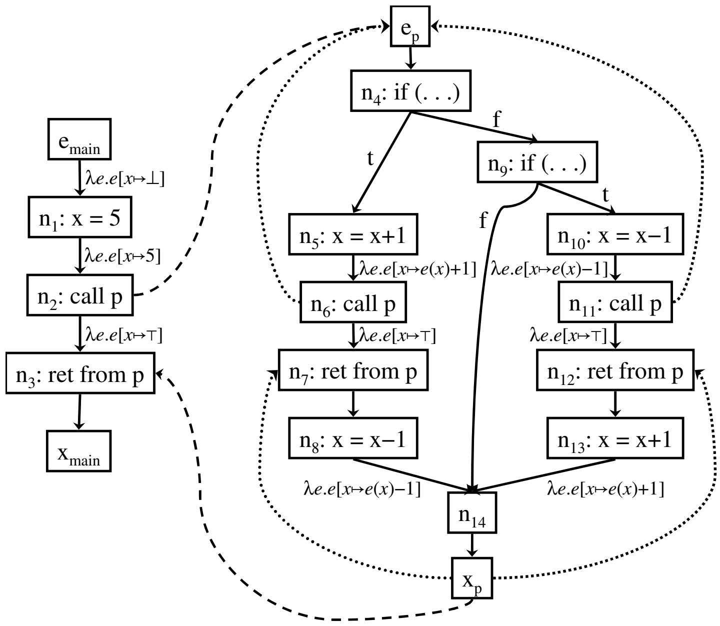


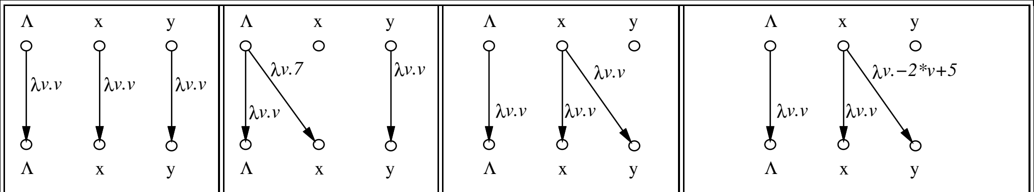
Fig. 21. A program fragment and its supergraph. The environment transformer for all unlabeled edges is $\lambda e.e$.

configurations. This will be illustrated by the examples in Section 4.3.3. Section 4.3.3 also shows how this approach provides a new algorithm for obtaining meet-over-all-paths dataflow information in a demand-driven fashion. Sections 4.3.1 and 4.3.2 lay the groundwork for these examples.

4.3.1 Background on "Interprocedural Distributive Environment" Problems

This section reviews the Interprocedural Distributive Environment (IDE) framework for context-sensitive interprocedural dataflow analysis [6]. This framework 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 domain of *values*) that has no infinite descending chains, 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 $y = -2 * x + 5$. Fig. 21 shows a program fragment and supergraph that will be used to illustrate an analysis for linear constant propagation. (To simplify the presentation, the environment transformer on the edge $e_{main} \to n_1$ is $\lambda e.v_0 \stackrel{\text{def}}{=} \lambda e.e[x \mapsto \bot]$.)



(a) $\lambda e.e$ (b) $\lambda e.e[x \mapsto 7]$ (c) $\lambda e.e[y \mapsto e(x)]$ (d) $\lambda e.e[y \mapsto -2 * e(x) + 5]$

Fig. 22. The exploded representations of four environment-transformer functions used in linear constant propagation.

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") [6]. Each micro-function on an edge $d_1 \to d_2$ 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. Fig. 22 shows the exploded representations of four environment-transformer functions used in linear constant propagation. Fig. 22(a) shows how the identity function $\lambda e.e$ is represented. Fig. 22(b)–Fig. 22(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 $\Lambda \to \Lambda$, labeled with $\lambda v.v$; these edges are needed to capture function composition properly [6].)

Fig. 23 shows the exploded supergraph that corresponds to the program from Fig. 21 for the linear constant-propagation problem.

4.3.2 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 elements are the functions

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

The elements of F_{lc} correspond to the micro-functions for linear constant propagation, closed under meet and composition.

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

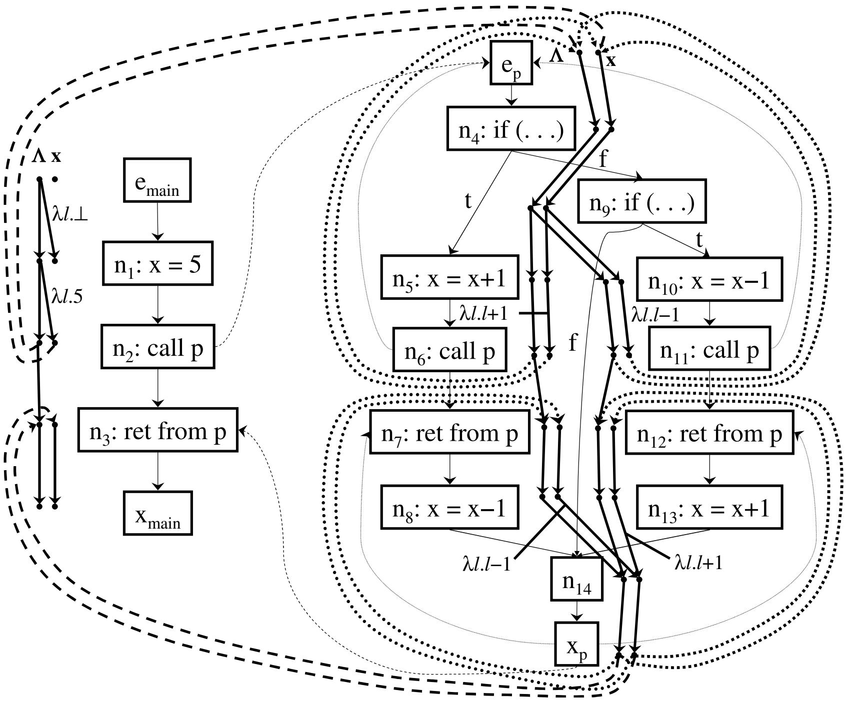


Fig. 23. The exploded supergraph of the program from Fig. 21 for the linear constant-propagation problem. The micro-functions are all *id*, except where indicated.

$a \in \mathbb{Z}$, $b \in \mathbb{Z}$, $c \in \mathbb{Z}_{\perp}^{\top}$, 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 ($\oplus$) of two functions can be represented. (See [6] 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, $\lambda l.\perp$ is represented by $(1, 0, \perp)$, and a constant function $\lambda l.b$, for $b \in \mathbb{Z}$, is represented as $(0, b, \top)$.

The operations $\oplus$ and $\otimes$ are meet and compose, respectively; they are defined as follows:

$$\lambda l.\top \oplus \lambda l.\top = \lambda l.\top$$
$$(a,b,c) \oplus \lambda l.\top = \lambda l.\top \oplus (a,b,c) = (a,b,c)$$
$$(a_2,b_2,c_2) \oplus (a_1,b_1,c_1) = \begin{cases} (a_1,b_1,c_1 \sqcap c_2) & \text{if } a_1 = a_2 \text{ and } b_1 = b_2 \\ (a_i,b_i,c) & \text{if } l_0 = (b_1 - b_2)/(a_2 - a_1) \in \mathbb{Z} \\ & \text{and } c_1 \sqcap c_2 \neq \bot, \text{where } a_i = \min(a_1,a_2) \\ & \text{and } c = (a_1 * l_0 + b_1) \sqcap c_1 \sqcap c_2, \\ (1,0,\bot) & \text{otherwise} \end{cases}$$

$$(a,b,c) \otimes \lambda l.\top = \lambda l.\top \otimes \lambda l.\top = \lambda l.\top \otimes (a,b,c) = \lambda l.\top$$
$$(a_2,b_2,c_2) \otimes (a_1,b_1,c_1) = ((a_1 * a_2),(a_1 * b_2 + b_1),((a_1 * c_2 + b_1) \sqcap c_1)).$$

Here it is assumed that $x * \top = \top * x = x + \top = \top + x = \top$ for $x \in \mathbb{Z}_\bot^\top$ and that $x * \bot = \bot * x = x + \bot = \bot + x = \bot$ for $x \in \mathbb{Z}_\bot$. The second case for $(a_2,b_2,c_2) \oplus (a_1,b_1,c_1)$ 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 encoding of the exploded supergraph as a weighted PDS is similar to the encoding of the unexploded supergraph described in Section 4.2, except that now there is a separate control location for each program variable (and also one for Λ). Stack symbols, such as n_4, n_5, and n_6, correspond to nodes of 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	$\langle x,n_5\rangle \hookrightarrow \langle x,n_6\rangle$	$\lambda l.l+1$

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

4.3.3 Obtaining Dataflow Information from the Weighted PDS of the Exploded Supergraph

For linear constant propagation, we are interested in a generalized pushdown 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 Alg. 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 4.3 *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	$\langle x, e_p\ n_3 \rangle$	*Accepted by query automaton*	

Example 4.4 *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 *(pushing* n_3*); p calls p at* n_6 *(pushing* n_7*); and finally p calls p at* n_{11} *(pushing* 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	$\langle x, e_p\ n_{12}\ n_7\ n_3\rangle$	*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.*

The next example represents a new algorithm for obtaining meet-over-all-paths dataflow information in a demand-driven fashion. (Previous work on demand-driven interprocedural dataflow analysis is discussed in Section 6.)

Example 4.5 *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 [21,6,19] 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
$\lambda l.\bot (= id \oplus \lambda l.l - 1)$	$\langle x, e_p\ n_3 \rangle$		
id	$\langle x, e_p\ n_3 \rangle$	*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	$\langle x, e_p\ n_{12}\ n_3 \rangle$	*Accepted by query automaton*	

4.3.4 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 3.1 and [15].

4.3.5 How Clients of Dataflow Analysis Can Take Advantage of this Machinery

Alg. 1 and the construction given above provide a new algorithm for interprocedural dataflow analysis. As demonstrated by Exs. 4.3, 4.4, and 4.5, 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:

- As demonstrated by Exs. 4.3, 4.4, and 4.5, a tool for program understanding could let users pose queries about dataflow information with respect to a regular language of initial stack configurations.
- 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.

The latter possibility is illustrated by Fig. 24, which shows a transformed version of the program from Fig. 21. The transformed program takes advantage of the information obtained from Ex. 4.3, namely, that in Fig. 21 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.

4.4 Extensions and Variations

Our definition of WPDSs imposes a distributivity property on semiring operations, i.e., for all $a, b, c \in D$,

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \qquad \text{and} \qquad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

(See Defn. 5(3).) This assumption restricts the dataflow-analysis problems that can be encoded with WPDSs to ones with distributive transfer functions. However, it is possible to loosen our requirements and replace Defn. 5(3) with

```
int x;

void main() {
  x = 5;
  p'();
  return;
}

void p() {
  if (...) {
     x = x + 1;
     p();
     x = x - 1;
  }
  else if (...) {
     x = x - 1;
     p();
     x = x + 1;
  }
  return;
}
```

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

Fig. 24. A transformed version of the program from Fig. 21 that takes advantage of the fact that in Fig. 21 the value of x is 5 whenever p is entered with a stack of the form "$e_p\ (n_{12}\ n_7)^*\ n_3$".

a weaker monotonicity condition: for all $a, b, c \in D$,

$$a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c) \qquad \text{and} \qquad (a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$$

This allows a broader class of dataflow-analysis problems to be treated.

The formalization is nearly the same as in the distributive case—i.e., using an automaton construction that is justified in terms of grammar flow analysis. The primary difference is that an iterative computation that finds the maximum-fixed-point value for each transition would, in general, provide only a *safe* solution rather than a precise one; that is, instead of the value $\delta(c)$ for a configuration c (which corresponds to the meet-over-all-derivations value for a particular grammar-flow-analysis problem), we would have to be content with a value that approximates ($\sqsubseteq$) $\delta(c)$. (This follows immediately from the material in Section 3, which formalizes GPP and GPS problems in terms of grammar flow analysis, and the theorems about the relationship between the maximum-fixed-point solution and the meet-over-all-derivations value for a monotonic grammar-flow-analysis problem [16,17].)

It would also be possible to abandon the "no-infinite-descending-chains" property, i.e., Defn. 5(5), and instead use widening to ensure that the automaton-construction algorithms terminate [25,26].

5 Differential Algorithms for Generalized Pushdown Reachability Problems

It has been observed that for some fixed-point-finding problems, it is possible to propagate changes in values ("deltas"), rather than full values [27]. (Subsequent work on differential fixed-point evaluation includes [28–30].)

Differential propagation can also be applied to GPR problems when the domain of weights supports an additional operator, denoted by $\dot{-}$, which returns a value that represents the difference between two values.[13] This section presents differential algorithms for GPP and GPS problems. (The two differential algorithms can also be extended to provide witness sets; however, these constructions will not be covered here.)

Differential dataflow-analysis algorithms can be seen as providing an adaptive strategy that fits in between algorithms such as the ones presented by Sharir and Pnueli [19] and Knoop and Steffen [21], and algorithms based on using an explicit exploded supergraph [32,6]. That is, when differential propagation is possible (i.e., when working with a weight domain for which a $\dot{-}$ operation is available), you do not have to perform an explicit "explosion" process. The specification can be written just as it would be for the application of a non-differential algorithm. Efficiency is obtained from differential propagation: rather than exploding every dataflow function ahead of time, which forces propagation to be carried out at a fine-grained level, the differential approach "peels off" an appropriate amount of work to propagate at each step. The amount peeled off can be as small as what one gets via explosion (i.e., propagation of one "factoid"), but when appropriate, larger collections of factoids will be propagated together [28].

[13] In an algebra in which you can put values back together using a meet operator, the proper term is "quotient operator"; when one can put values back together with join, then one has a "difference operator" (e.g., see [31]). However, "difference operator" is a more suggestive term (i.e., in terms of providing intuition about the role of $\dot{-}$). In any case, by duality, a bounded idempotent semiring could be considered to be a join semilattice, rather than a meet semilattice. Thus, we will continue to refer to $\dot{-}$ as a difference operator.

However, the reader should be aware that because the formalization is based on meet semilattices, we have $a \dot{-} b \sqsupseteq a$ (i.e., the difference of a and b is *greater than or equal to* a, which may be counterintuitive).

We assume that, for all a, b, and c, the following properties hold for $\dot{-}$:

$$a \oplus (b \mathbin{\dot{-}} a) = a \oplus b \tag{3}$$
$$(a \mathbin{\dot{-}} b) \mathbin{\dot{-}} c = (a \mathbin{\dot{-}} (b \oplus c) \tag{4}$$
$$a \sqsubseteq b \Leftrightarrow b \mathbin{\dot{-}} a = 0 \tag{5}$$

Several other properties hold as consequences of Eqns. (3)–(5):

$$a \mathbin{\dot{-}} a = 0 \tag{6}$$
$$b \mathbin{\dot{-}} a \sqsupseteq b \tag{7}$$
$$a \sqsubseteq b \Rightarrow a \mathbin{\dot{-}} c \sqsubseteq b \mathbin{\dot{-}} c \quad \text{monotonicity in } 1^{st} \text{ argument} \tag{8}$$
$$(a \mathbin{\dot{-}} b) \mathbin{\dot{-}} b = a \mathbin{\dot{-}} b \quad \text{idempotence} \tag{9}$$

Proofs of Eqns. (6)–(9) are given in Appendix A.

5.1 Differential Weighted pre*

Fig. 25 presents a differential algorithm for creating a weighted automaton for a GPP problem. The algorithm is quite similar to Alg. 1 from Fig. 9; the differences between the two algorithms, which are indicated with underlining in Fig. 25, are as follows:

- Instead of using an explicit workset to record the transitions whose values have changed, in Alg. 5 there is an additional value $\delta l(t)$ associated with each transition t; $\delta l(t)$ holds the accumulated change in t's value since it was most recently considered (see lines 4, 5, 12, and 13). On each iteration of the loop body, a transition t with a non-0 value of $\delta l(t)$ is considered (see line 12).
- In lines 14, 16, and 18, a *change* in a transition's weight is calculated using an expression of the form $\ldots \otimes \mathit{delta} \otimes \ldots$
- Whenever an $\mathit{update}(t, \mathit{del})$ is performed, $\delta l(t)$ is updated with the value $\mathit{del} \mathbin{\dot{-}} l(t)$ (see line 4).

(See Ex. 5.1 for a concrete example that illustrates the related algorithm for differential weighted *post**.)

We now wish to argue that Alg. 5 solves the GPP problem. In Section 3.1, we argued that Alg. 1 solves the GPP problem because it solves the abstract grammar problem for the productive *PopSeq* nonterminals in the grammar from Fig. 6; Alg. 1 finds the maximum fixed-point solution, which by Theorem 13 equals the desired meet-over-all-derivations value. The correctness of Alg. 5 follows from a similar argument: it solves the GPP problem because it also solves the abstract grammar problem for the productive *PopSeq* nonterminals in the grammar from Fig. 6—in essence, using a differential algorithm

Algorithm 5
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
$m_G(PopSeq_{(q,\gamma,q')})$ in the abstract grammar problem defined in Fig. 6.

procedure $update(t, \underline{del})$
begin
  $\to := \to \cup \{t\}$
  $\underline{\delta l(t) := \delta l(t) \oplus (del \mathbin{\dot{-}} l(t))}$
  $\underline{l(t) := l(t) \oplus del}$
end

$\to := \to_0$; $\underline{\delta l := \lambda t.0}$; $l := \lambda t.0$
for all $t \in \to_0$ **do** $\underline{\delta l(t) := 1}$; $l(t) := 1$
for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $update((p, \gamma, p'), f(r))$
while $\underline{\text{there exists } t \text{ such that } \delta l(t) \neq 0}$ **do**
  select $\underline{\text{a } t = (q, \gamma, q') \text{ such that } \delta l(t) \neq 0}$
  $\underline{delta := \delta l(t);\ \delta l(t) := 0}$
  for all $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in \Delta$ **do** $update((p_1, \gamma_1, q'), f(r) \otimes \underline{delta})$
  for all $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta$ **do**
    for all $t' = (q', \gamma_2, q'') \in \to$ **do** $update((p_1, \gamma_1, q''), f(r) \otimes \underline{delta} \otimes l(t'))$
  for all $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p', \gamma_2\gamma \rangle \in \Delta$ **do**
    if $t' = (p', \gamma_2, q) \in \to$ **then** $update((p_1, \gamma_1, q'), f(r) \otimes l(t') \otimes \underline{delta})$
return $((Q, \Gamma, \to, P, F), l)$

Fig. 25. A differential algorithm for creating a weighted automaton for the weighted *pre** problem. The algorithm is quite similar to Alg. 1 from Fig. 9; the differences between the two algorithms are indicated with underlining.

for solving grammar-flow-analysis problems. The fact that a differential algorithm can be used to solve grammar-flow-analysis problems follows from previous work on differential algorithms for finding fixed points [27,28]. To make the paper self-contained, and to elucidate more clearly the connection between Algs. 5 and 6 and differential fixed-point finding for grammar-flow analysis, Appendix B gives a specific differential algorithm for grammar flow analysis (Alg. 8), and proves that the algorithm finds the maximum fixed-point solution (or, equivalently, the meet-over-all-derivations solution).

*5.2 Differential Weighted post**

Fig. 26 presents a differential algorithm for creating a weighted automaton for the weighted *post** problem. Again, the algorithm is quite similar to its

Algorithm 6

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 $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ that accepts C, such that
$\mathcal{A}$ has no transitions into P states and has no ε-transitions.

Output: a $\mathcal{P}$-automaton $\mathcal{A}_{post^*} = (Q', \Gamma, \rightarrow, P, F)$ that accepts $post^*(C)$;
a function l that maps every $(q, \gamma, q') \in \rightarrow$ to the value of
$m_G(PushSeq_{(q,\gamma,q')})$ in the abstract grammar problem defined in Fig. 16.

procedure $update(t, \underline{del})$
begin
  $\rightarrow := \rightarrow \cup \{t\}$
  $\underline{\delta l(t) := \delta l(t) \oplus (del \mathbin{\dot{-}} l(t))}$
  $\underline{l(t) := l(t) \oplus del}$
end

$\underline{D}$ $update'(t, \underline{del})$
begin
  $\rightarrow := \rightarrow \cup \{t\}$
  $\underline{delta := del \mathbin{\dot{-}} l(t)}$
  $\underline{l(t) := l(t) \oplus del}$
  return $\underline{delta}$
end

$\rightarrow := \rightarrow_0$; $\underline{\delta l := \lambda t.0}$; $l := \lambda t.0$
for all $t \in \rightarrow_0$ **do** $\underline{\delta l(t) := 1}$; $l(t) := 1$
$Q' := Q$; **for all** $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$ **do** $Q' := Q' \cup \{q_{p',\gamma'}\}$
while $\underline{\text{there exists } t \text{ such that } \delta l(t) \neq 0}$ **do**
  select $\underline{\text{a } t = (p, \gamma, q) \text{ such that } \delta l(t) \neq 0}$
  $\underline{delta := \delta l(t);\ \delta l(t) := 0}$
  if $\gamma \neq \varepsilon$ **then**
    for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ **do** $update((p', \varepsilon, q), \underline{delta} \otimes f(r))$
    for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$ **do** $update((p', \gamma', q), \underline{delta} \otimes f(r))$
    for all $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \Delta$ **do**
      $update((p', \gamma', q_{p',\gamma'}), 1)$
      $\underline{delta'} := update'((q_{p',\gamma'}, \gamma'', q), \underline{delta} \otimes f(r))$
      if $\underline{delta' \neq 0}$ **then**
        for all $t' = (p'', \varepsilon, q_{p',\gamma'})$ **do** $update((p'', \gamma'', q), \underline{delta'} \otimes l(t'))$
  else
    for all $t' = (q, \gamma', q') \in \rightarrow$ **do** $update((p, \gamma', q'), l(t') \otimes \underline{delta})$
return $((Q', \Gamma, \rightarrow, P, F), l)$

Fig. 26. A differential algorithm for creating a weighted automaton for the weighted $post^*$ problem. The algorithm is quite similar to Alg. 3 from Fig. 17; the differences between the two algorithms are indicated with underlining.

non-differential counterpart (i.e., Alg. 3 from Fig. 17); the differences between the two algorithms are indicated with underlining in Fig. 26.

The ideas used in Fig. 26 are similar to the ones used in Fig. 25:

- Instead of an explicit workset of transitions whose values have changed, Alg. 5 maintains an accumulated change in value, $\delta l(t)$, for each transition t. On each iteration of the loop body, a transition t with a non-0 value of $\delta l(t)$ is considered.
- A change in a transition's weight is calculated using an expression of the form $\ldots \otimes \mathit{delta} \otimes \ldots$
- Whenever an $\mathit{update}(t, \mathit{del})$ is performed, $\delta l(t)$ is updated with the value $\mathit{del} \dot{-} l(t)$.

Example 5.1 *To demonstrate Alg. 6, we extend the constant-propagation example from Ex. 4.2 with a difference operator, $a \dot{-} b$, which returns a restricted to those variable updates where a and b differ. The operators $\otimes$ and $\oplus$ are modified to handle partial environment transformers: $a \otimes b$ has as its domain the intersection of the domains of a and b; $a \oplus b$ has as its domain the union of the domains of a and b. If the domain of a (b) does not include program variable x in the domain of b (a), then $a \oplus b$ takes its mapping update for x from b (a).*

The example discussed in Ex. 4.2 illustrates the benefits of the differential algorithm, because slightly different environment transformers are computed along the two branches of the if statement (see Fig. 18). The environment transformer for the path to n_8 that takes the false branch of the if statement is $\lambda e.e[g \mapsto 0, h \mapsto 0]$. When the true branch is taken, the environment transformer for the path to n_8 is $\lambda e.e[g \mapsto 1, h \mapsto 0]$. Both the differential and non-differential algorithms may propagate (at different stages) two different update values for g: 0 or 1, and $\perp$. If the non-differential algorithm propagates two update values for g, it must also propagate the update value 0 for h twice; the differential algorithm will only propagate the update value 0 for h once.

Consider an execution of Alg. 6 where the sequence of transitions selected by line 20 is $(p, \mathit{entry}_{main}, q)$, (p, n_1, q), (p, n_2, q), $\ldots$, (p, n_4, q), and corresponds to the path through the false branch of the if statement. At this point, we have $\delta l((p, n_8, q)) = l((p, n_8, q)) = \lambda e.e[g \mapsto 0, h \mapsto 0]$. Suppose that (p, n_8, q) is the next transition selected at line 20; the entire environment $\lambda e.e[g \mapsto 0, h \mapsto 0]$ is propagated to $\delta l(p, \mathit{exit}_{main}, q)$ and $l(p, \mathit{exit}_{main}, q)$ by the call to update on line 24.

Eventually, Alg. 6 will process $t = (p, n_7, q)$ with $\delta l(t) = l(t) = \lambda e.e[g \mapsto 1, h \mapsto 0]$. Because of the rule $\langle p, n_7 \rangle \hookrightarrow \langle p, n_8 \rangle$, this results in the call $\mathit{update}((p, n_8, q), \lambda e.e[g \mapsto 1, h \mapsto 0]$ (on line 24), which yields $\delta l((p, \mathit{exit}_{main}, q))$ $= \lambda e \downarrow_g .e[g \mapsto \perp]$ (where $e \downarrow_g$ denotes e restricted to $\{g\}$) and $l((p, \mathit{exit}_{main}, q))$

Algorithm 7
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 $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ that accepts C, such that
$\mathcal{A}$ has no transitions into P states and has no ε-transitions.

Output: for each $p \in P$ and $\gamma \in \Gamma$, compute $V_{p,\gamma} := \bigoplus\{\, \delta(\langle p, \gamma w\rangle) \mid w \in \Gamma^* \,\}$.

Let $\mathcal{A}_{post^*} = (Q', \Gamma, \rightarrow, P, F)$ and l be the output from calling Alg. 3 or Alg. 6

for all $q \in Q' \setminus (P \cup F)$ **do** $\underline{\delta W_q := 0}$; $W_q := 0$
for all $q \in F$ **do** $\underline{\delta W_q := 1}$; $W_q := 1$
while $\underline{\text{there exists } q \text{ such that } \delta W_q \neq 0}$ **do**
  $\underline{\text{select a } q \text{ such that } \delta W_q \neq 0}$
  $\underline{\mathit{delta} := \delta W_q;\ \delta W_q := 0}$
  for all $t = (q', \gamma, q) \in \rightarrow,\ q' \notin P$ **do**
    $\underline{\delta W_{q'} := \delta W_{q'} \oplus ((\mathit{delta} \otimes l(t)) \mathbin{\dot{-}} W_{q'})}$
    $\underline{W_{q'} := W_{q'} \oplus (\mathit{delta} \otimes l(t))}$

for all $p \in P$ and $\gamma \in \Gamma$ **do** $V_{p,\gamma} := \bigoplus_{t=(p,\gamma,q)\in\rightarrow}(W_q \otimes l(t))$

Fig. 27. A differential algorithm that, for each $p \in P$ and $\gamma \in \Gamma$, computes $V_{p,\gamma}$. The algorithm is quite similar to Alg. 4 from Fig. 19; the differences between the two algorithms are indicated with underlining.

$= \lambda e.e[g \mapsto 0, h \mapsto 0].$

When transition (p, n_8, q) is next selected for processing (on line 20), only the update for g will be propagated to $\delta l(p, exit_{main}, q)$ and $l(p, exit_{main}, q)$. In contrast, the non-differential algorithm would also propagate the update for h, even though the update for h has not changed since the last time that (p, n_8, q) was processed. (Note, the realized savings in the cost of propagating the new environment transformer would be much greater if there were more nodes reachable from n_8.)

The correctness of Alg. 6 follows from the same argument that was used to justify Alg. 5 in Section 5.1: Alg. 6 solves the GPS problem because it solves the abstract grammar problem for the productive *PushSeq* and *SameLevelSeq* nonterminals in the grammar from Fig. 15; Alg. 6 uses a differential algorithm for solving grammar-flow-analysis problems, which is justified by the material on differential propagation for grammar-flow analysis that is presented in Appendix B.

5.3 A Differential Algorithm for Obtaining Conventional Dataflow Information

The goal of conventional dataflow analysis is to obtain the meet-over-all-valid-paths value for each node in the supergraph. As discussed in Section 4.2, these values can be obtained by formulating such dataflow problems in the GPP setting and using the fixed-point computation from Alg. 4, which computes, for every supergraph node n,

$$\mathrm{MOVP}_n = V_{p,\gamma_n} = \bigoplus\{\, \delta(\langle p, \gamma_n w\rangle) \mid w \in \Gamma^* \,\}.$$

To recap, Alg. 4 computes W_q for every non-initial state q, where W_q is the value obtained by following all accepting paths starting at q, accumulating semiring values for each path, and taking the meet over all these values. This is achieved by propagating values backwards over transitions of $\mathcal{A}_{post^*}$.

In this respect, Alg. 4 is formulated in the same spirit as the non-differential *post** algorithm from Fig. 17: whenever W_q changes, Alg. 4 recomputes the values of the predecessor states of q, using the full new value of W_q.

Alg. 7 in Fig. 27 presents an alternative method that can be used when a $\dot{-}$ operation is available. Phases 1 and 3 (lines 1 and 12) are the same as in Alg. 4; the differences in phase 2 are as follows:

- Instead of maintaining a workset, Alg. 7 maintains a value δW_q for every q, which reflects the accumulated change in W_q since the last time q was selected in line 6. In lines 9 and 10, this value is used instead of W_q to update the predecessor states of q.
- A minor change is that (due to the previous change), in every iteration of the while-loop, the values of the *predecessors* of the selected state q are updated, instead of W_q itself.

Note that the construction of the automaton and the computation of the $V_{p,\gamma}$ values are not interleaved. In principle, either of Algs. 4 and 7 can be applied to an automaton obtained from either of Algs. 3 and 6, although Algs. 6 and 7 can be applied only when a $\dot{-}$ operation is available.

(The comments from Section 4.2 about how to carry out a similar computation on a *pre** automaton apply here, as well.)

6 Related Work

Several connections between dataflow analysis and model checking have been established in past work [4,5,3,1]. 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 [13–15]—but generalized from its original form [8]—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 more basic model-checking primitives, namely, pre^* and $post^*$.

These ideas have been illustrated by applying them to simple constant propagation and to linear constant propagation; in particular, linear constant propagation is not expressible as a bit-vector problem.

Moped [22] is a system for PDS model checking that supports the ability to model variables of finite data types (e.g., Booleans and integers with values drawn from a finite range). Moped can be viewed as an implementation of a particular class of WPDSs in which weights are limited to finite-domain relations. In contrast, this paper allows weights to be drawn from a bounded idempotent semiring, which means that its results apply to dataflow-analysis problems that involve certain classes of abstract domains that have infinite cardinality; such problems include linear constant propagation [6] and affine-relation analysis [7].

Bouajjani, Esparza, and Toulli [33] 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 differs significantly from ours. Instead of deriving the weights directly, they are obtained using a fixed-point 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 "no-infinite-descending-chains" 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 [33] have only finite descending chains, so Alg. 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 [34,35] 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" [36], 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 [37], 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 a class of dataflow-analysis problems called the IFDS problems [32].[14] They also gave an explicit demand algorithm for IFDS problems that does not rely on the magic-sets transformation [38].
- Both exhaustive and demand algorithms for solving a certain class of IDE problems are presented in [6]. The relationship between the two algorithms given in that paper is similar to the relationship between the exhaustive [32] and demand [38] algorithms for IFDS problems.
- A fourth approach to obtaining demand versions of interprocedural dataflow-analysis algorithms was investigated by Duesterwald, Gupta, and Soffa [39]. 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 [13–15] on the graphs used in [32,38]; 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

[14] Logic-programming terminology is not used in [32]; 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.

to a language of stack configurations, using our recent generalization of PDS reachability algorithms to weighted PDSs [8].

It should be noted that, like the algorithms from [6], the algorithms for solving GPR problems given in Section 3 are not guaranteed to terminate for all IDE problems; however, like the algorithms from [6], they do 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 [6]; however, that algorithm is strictly less general than the algorithm presented here (cf. Ex. 4.5).

An application of the theory of PDSs to interprocedural dataflow analysis has been proposed by Esparza and Knoop [2], who considered several bit-vector problems, as well as the faint-variables problem, which is an IFDS problem [40, 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.)

Model checking of PDSs has previously been used for verifying security properties of programs [41–43]. 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.

The methods described in this paper have been used to create implementations of context-sensitive interprocedural dataflow analyses for uninitialized variables, live variables, linear constant propagation, and the detection of affine relationships. The most ambitious of these is the implementation of affine-relation analysis. This implements a solver for an interprocedural dataflow-analysis problem due to Müller-Olm and Seidl [7], which determines, for each program point n, the set of all affine relations that hold among program variables whenever n is executed. This method can be re-cast as solving a GPR problem (and solved with the same asymptotic complexity). Balakrishnan and Reps have created such an implementation using the WPDS++ library [10]. This is used as one of several analysis phases in a system for analyzing memory accesses in x86 executables to recover intermediate representations that are similar to those that can be created for a program written in a high-level language [44].

Acknowledgments

We thank H. Seidl for making available an early version of [7]. We have also benefited from the observations of G. Balakrishnan, N. Kidd, and A. Lal.

References

[1] P. Cousot, R. Cousot, Temporal abstract interpretation, in: Symp. on Princ. of Prog. Lang., 2000, pp. 12–25.

[2] J. Esparza, J. Knoop, An automata-theoretic approach to interprocedural data-flow analysis, in: Found. of Softw. Sci. and Comp. Structures, Vol. 1578 of LNCS, Springer, 1999, pp. 14–30.

[3] D. Schmidt, Data-flow analysis is model checking of abstract interpretations, in: Symp. on Princ. of Prog. Lang., ACM Press, New York, NY, 1998, pp. 38–48.

[4] B. Steffen, Data flow analysis as model checking, in: Int. Conf. on Theor. Aspects of Comp. Softw., Vol. 526 of Lec. Notes in Comp. Sci., Springer-Verlag, 1991, pp. 346–365.

[5] B. Steffen, Generating data flow analysis algorithms from modal specifications, Sci. of Comp. Prog. 21 (2) (1993) 115–139.

[6] M. Sagiv, T. Reps, S. Horwitz, Precise interprocedural dataflow analysis with applications to constant propagation, Theor. Comp. Sci. 167 (1996) 131–170.

[7] M. Müller-Olm, H. Seidl, Precise interprocedural analysis through linear algebra, in: Symp. on Princ. of Prog. Lang., 2004.

[8] S. Schwoon, S. Jha, T. Reps, S. Stubblebine, On generalized authorization problems, in: Comp. Sec. Found. Workshop, IEEE Comp. Soc., Wash., DC, 2003.

[9] S. Schwoon, WPDS: A library for weighted pushdown systems, http://www.fmi.uni-stuttgart.de/szs/tools/wpds/ (2003).

[10] N. Kidd, T. Reps, D. Melski, A. Lal, WPDS++: A C++ library for weighted pushdown systems, http://www.cs.wisc.edu/wpis/wpds++/ (2004).

[11] S. Schwoon, Model-checking pushdown systems, Ph.D. thesis, Technical Univ. of Munich, Munich, Germany (July 2002).

[12] S. Jha, T. Reps, Analysis of SPKI/SDSI certificates using model checking, in: IEEE Comp. Sec. Found. Workshop (CSFW), IEEE Computer Society Press, 2002.

[13] A. Bouajjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: Application to model checking, in: Proc. CONCUR, Vol. 1243 of Lec. Notes in Comp. Sci., Springer-Verlag, 1997, pp. 135–150.

[14] A. Finkel, B.Willems, P. Wolper, A direct symbolic approach to model checking pushdown systems, Electronic Notes in Theoretical Comp. Sci. 9.

[15] J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon, Efficient algorithms for model checking pushdown systems, in: Proc. Computer-Aided Verif., Vol. 1855 of Lec. Notes in Comp. Sci., 2000, pp. 232–247.

[16] U. Moencke, R. Wilhelm, Grammar flow analysis, in: H. Alblas, B. Melichar (Eds.), Attribute Grammars, Applications and Systems, Vol. 545 of Lec. Notes in Comp. Sci., Springer-Verlag, Prague, Czechoslovakia, 1991, pp. 151–186.

[17] G. Ramalingam, Bounded Incremental Computation, Vol. 1089 of Lec. Notes in Comp. Sci., Springer-Verlag, 1996.

[18] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1985.

[19] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: S. Muchnick, N. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981, Ch. 7, pp. 189–234.

[20] G. Kildall, A unified approach to global program optimization, in: Symp. on Princ. of Prog. Lang., ACM Press, New York, NY, 1973, pp. 194–206.

[21] J. Knoop, B. Steffen, The interprocedural coincidence theorem, in: Int. Conf. on Comp. Construct., 1992, pp. 125–140.

[22] S. Schwoon, Moped: A model-checker for pushdown systems, http://www.fmi.uni-stuttgart.de/szs/tools/moped/ (2002).

[23] A. Lal, T. Reps, G. Balakrishnan, Extended weighted pushdown systems, in: Proc. Computer-Aided Verif., 2005.

[24] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, 1999.

[25] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Symp. on Princ. of Prog. Lang., ACM Press, New York, NY, 1979, pp. 269–282.

[26] F. Bourdoncle, Efficient chaotic iteration strategies with widenings, in: Int. Conf. on Formal Methods in Prog. and their Appl., Lec. Notes in Comp. Sci., Springer-Verlag, 1993.

[27] J. Cai, R. Paige, Program derivation by fixed point computation, Sci. of Comp. Program. 11 (3) (1989) 197–261.

[28] C. Fecht, H. Seidl, Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems, Nordic J. of Comput. (5) (1998) 304–329.

[29] J. Ahn, A differential evaluation of fixpoint iterations, in: Asian Workshop on Prog. Lang. and Syst., 2001.

[30] H. Eo, K. Yi, An improved differential fixpoint iteration method for program analysis, in: Asian Workshop on Prog. Lang. and Syst., 2002.

[31] T. Reps, Algebraic properties of program integration, Sci. of Comp. Program. 17 (1991) 139–215.

[32] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Symp. on Princ. of Prog. Lang., ACM Press, New York, NY, 1995, pp. 49–61.

[33] A. Bouajjani, J. Esparza, T. Touili, A generic approach to the static analysis of concurrent programs with procedures, in: Proc. Symp. on Princ. of Prog. Lang., 2003, pp. 62–73.

[34] T. Reps, Solving demand versions of interprocedural analysis problems, in: P. Fritzson (Ed.), Proceedings of the Fifth International Conference on Compiler Construction, Vol. 786 of Lec. Notes in Comp. Sci., Springer-Verlag, Edinburgh, Scotland, 1994, pp. 389–403.

[35] T. Reps, Demand interprocedural program analysis using logic databases, in: R. Ramakrishnan (Ed.), Applications of Logic Databases, Kluwer Academic Publishers, 1994.

[36] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman, Magic sets and other strange ways to implement logic programs, in: Proceedings of the Fifth ACM Symposium on Principles of Database Systems, ACM Press, New York, NY, 1986.

[37] D. Warren, Memoing for logic programs, Communications of the ACM 35 (3) (1992) 93–111.

[38] S. Horwitz, T. Reps, M. Sagiv, Demand interprocedural dataflow analysis, in: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, New York, NY, 1995, pp. 104–115.

[39] E. Duesterwald, R. Gupta, M. Soffa, Demand-driven computation of interprocedural data flow, in: Symp. on Princ. of Prog. Lang., ACM Press, New York, NY, 1995, pp. 37–48.

[40] T. Reps, M. Sagiv, S. Horwitz, Interprocedural dataflow analysis via graph reachability, Tech. Rep. TR 94-14, Datalogisk Institut, Univ. of Copenhagen, available at "http://www.cs.wisc.edu/wpis/papers/diku-tr94-14.ps" (1994).

[41] J. Esparza, A. Kučera, S. Schwoon, Model-checking LTL with regular valuations for pushdown systems, in: Proceedings of TACAS'01, Vol. 2031 of LNCS, Springer, 2001, pp. 306–339.

[42] T. Jensen, D. Le Metayer, T. Thorn, Verification of control flow based security properties, in: 1999 IEEE Symposium on Security and Privacy, 1999.

[43] H. Chen, D. Wagner, MOPS: An infrastructure for examining security properties of software, in: Conf. on Comp. and Commun. Sec., 2002.

[44] G. Balakrishnan, T. Reps, Analyzing memory accesses in x86 executables, in: Int. Conf. on Comp. Construct., 2004.

A Properties of the $\dot{-}$ Operation

Eqn. (6): Show that $a \dot{-} a = 0$

$$\begin{aligned} a &\sqsubseteq a \\ \therefore a \dot{-} a &= 0 && \text{by Eqn. (5)} \end{aligned}$$

Eqn. (7): Show that $b \dot{-} a \sqsupseteq b$

$$\begin{aligned} (b \dot{-} a) \dot{-} b &= b \dot{-} (a \oplus b) && \text{by Eqn. (4)} \\ &= (b \dot{-} b) \dot{-} a && \text{by Eqn. (4)} \\ &= 0 \dot{-} a && \text{by Eqn. (6)} \\ &= 0 && \text{by Eqn. (5)} \\ \therefore b \dot{-} a &\sqsupseteq b && \text{by Eqn. (5)} \end{aligned}$$

Eqn. (8): Show that $a \sqsubseteq b \Rightarrow a \dot{-} c \sqsubseteq b \dot{-} c$

Suppose that $a \sqsubseteq b$.

$$\begin{aligned} (b \dot{-} c) \dot{-} (a \dot{-} c) &= b \dot{-} (c \oplus (a \dot{-} c)) && \text{by Eqn. (4)} \\ &= b \dot{-} (c \oplus a) && \text{by Eqn. (3)} \\ &= (b \dot{-} a) \dot{-} c && \text{by Eqn. (4)} \\ &= 0 \dot{-} c && \text{by Eqn. (5)} \\ &= 0 && \text{by Eqn. (5)} \\ \therefore (a \dot{-} c) &\sqsubseteq (b \dot{-} c) && \text{by Eqn. (5)} \end{aligned}$$

Eqn. (9): Show that $(a \dot{-} b) \dot{-} b = a \dot{-} b$

$$\begin{aligned} (a \dot{-} b) \dot{-} b &= a \dot{-} (b \oplus b) && \text{by Eqn. (4)} \\ &= a \dot{-} b \end{aligned}$$

B Differential Grammar-Flow Analysis

A grammar-flow-analysis problem gives rise to a set of equations over the variables $\{V[X] \mid X \text{ a nonterminal}\}$. For brevity, we will also denote the variable set by $\overrightarrow{V}$. Each equation in the grammar-flow-analysis problem is of the form

$$V[X] = \bigoplus_{X \to g(X_1,\ldots,X_k) \in \text{Prods}} g(V[X_1], \ldots, V[X_k]). \tag{B.1}$$

Eqn. (B.1) can be summarized as

$$\overrightarrow{V} = G(\overrightarrow{V}), \tag{B.2}$$

where G denotes the collection of right-hand sides from Eqn. (B.1). One non-differential algorithm for finding the maximum fixed point is to form the iteration sequence

$$\begin{aligned} \overrightarrow{V_0} &= \overrightarrow{0} \\ \overrightarrow{V_{i+1}} &= G(\overrightarrow{V_i}) \end{aligned}$$

Therefore, by Theorem 13 ("the maximum fixed point equals the meet-over-all-derivations value"),

$$\overrightarrow{m_G} = \bigoplus_{i=0}^{\infty} G^i(\overrightarrow{V_0}).$$

To compare the answer produced by this iteration sequence with the one produced by the differential algorithm presented below, it is useful to observe that the exact same sequence of $\overrightarrow{V_i}$ is produced by the iteration sequence defined by

$$\begin{aligned} \overrightarrow{V_0} &= \overrightarrow{0} \\ \overrightarrow{V_{i+1}} &= \overrightarrow{V_i} \oplus G(\overrightarrow{V_i}) \end{aligned}$$

(A simple inductive argument establishes that for all $i \geq 0$, $\overrightarrow{V_i} \sqsupseteq G(\overrightarrow{V_i})$; hence, $\overrightarrow{V_i} \oplus G(\overrightarrow{V_i}) = G(\overrightarrow{V_i})$, and thus $\overrightarrow{V_{i+1}} = G(\overrightarrow{V_i})$.)

We define the operator $H(\overrightarrow{V}) \stackrel{\text{def}}{=} \overrightarrow{V} \oplus G(\overrightarrow{V})$. The maximum fixed point of H equals the maximum fixed point of G; thus, Theorem 13 implies that

$$\overrightarrow{m_G} = \bigoplus_{i=0}^{\infty} H^i(\overrightarrow{V_0}).$$

To perform differential propagation, we will use two vectors of variables, $\overrightarrow{V}$ and $\overrightarrow{\delta V}$. A differential algorithm for grammar flow analysis is presented as Alg. 8 of Fig. B.1. By design, it is similar in style to Algs. 5 and 6 of Section 5.

We now prove several lemmas that establish certain properties of Alg. 8. The first lemma shows that $\delta V[X]$ captures a part of $V[X]$'s value.

Lemma 21 *Each time control reaches line 3 of Alg. 8, $\overrightarrow{\delta V} \sqsupseteq \overrightarrow{V}$ (i.e., for all $X \in NT$, $\delta V[X] \sqsupseteq V[X]$).*

PROOF. The proof is by induction on the number of times that control reaches line 3.

Base case: The property is established by lines 1 and 2.

Algorithm 8
Input: an abstract grammar over $(S, \oplus)$, with nonterminals NT and productions Prods

Output: a value in S for each $X \in$ NT

1 $\overrightarrow{\delta V} := \overrightarrow{0}$; $\overrightarrow{V} := \overrightarrow{0}$
2 **for all** $X \in$ NT **do** $\delta V[X] := \bigoplus_{X \to g() \in \text{Prods}} g()$; $V[X] := \bigoplus_{X \to g() \in \text{Prods}} g()$
3 **while** there exists $Y \in$ NT such that $\delta V[Y] \neq 0$ **do**
4   select a Y such that $\delta V[Y] \neq 0$
5   $delta := \delta V[Y]$; $\delta V[Y] := 0$
6   **for all** $X \to g(X_1, \ldots, Y, \ldots, X_k) \in$ Prods **do**
7     $\delta V[X] := \delta V[X] \oplus (g(V[X_1], \ldots, delta, \ldots, V[X_k]) \mathbin{\dot{-}} V[X])$
8     $V[X] := V[X] \oplus g(V[X_1], \ldots, delta, \ldots, V[X_k])$
9 **return** $\overrightarrow{V}$

Fig. B.1. A differential algorithm for grammar flow analysis. (In line 2, "$X \to g()$" means a production with zero nonterminals on the right-hand side.)

Induction step: Assume that the property holds the i^{th} time that control reaches line 3. The property holds the $(i+1)^{st}$ time that control reaches line 3 because the property is maintained by lines 7 and 8: comparing the corresponding arguments of $\oplus$ on the right-hand sides of lines 7 and 8, we have

- $\delta V[X] \sqsupseteq V[X]$, by the induction hypothesis, and
- $(g(V[X_1], \ldots, delta, \ldots, V[X_k]) \mathbin{\dot{-}} V[X]) \sqsupseteq g(V[X_1], \ldots, delta, \ldots, V[X_k])$, by Eqn. (7).

Hence, by the fact that $\oplus$ is monotonic in each argument, the value of $\delta V[X]$ after line 7 is $\sqsupseteq$ the value of $V[X]$ after line 8.

□

To be able to discuss how values are propagated by Alg. 8, we introduce the following notation:

Definition 22 *For all $X \in NT$, at each moment during a run of Alg. 8, let $\overline{V[X]}$ denote the value that $V[X]$ had when X was most recently selected at line 4. If X has never been selected at line 4, $\overline{V[X]}$ is 0.*

The next lemma shows that $\delta V[X]$ captures the accumulated change in $V[X]$'s value since it was most recently selected at line 4.

Lemma 23 *For all $X \in NT$, each time that control reaches line 6 of Alg. 8, $\delta V[X] \oplus \overline{V[X]} = V[X]$.*

PROOF. The proof is by induction on the number of times that control reaches line 6.

Base case: The first time that control reaches line 6, either X has never been selected or the nonterminal in question is Y. In the former case, $\overline{V[X]} = 0$, $\delta V[X] = V[X]$, and thus the desired property holds; in the latter case, $\overline{V[Y]} = V[Y]$, $\delta V[Y] = 0$, and the property also holds.

Induction step: Assume that the property holds the i^{th} time that control reaches line 6. In general, the loop on lines 6–8 adjusts the values of $\delta V[X]$ and $V[X]$, for many different X's. We now show that each execution of the pair of assignments in lines 7 and 8 preserves the desired property. Let $\delta V[X]'$ and $V[X]'$ denote the values established in lines 7 and 8, respectively.

$$\delta V[X]' = \delta V[X] \oplus (g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \dot{-} V[X])$$

$$\begin{aligned}
&\delta V[X]' \oplus \overline{V[X]} \\
&\quad = \delta V[X] \oplus (g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \dot{-} V[X]) \oplus \overline{V[X]} \\
&\quad = \delta V[X] \oplus (g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \dot{-} (\delta V[X] \oplus \overline{V[X]})) \oplus \overline{V[X]} && \text{by the ind. hyp.} \\
&\quad = \delta V[X] \oplus ((g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \dot{-} \delta V[X]) \dot{-} \overline{V[X]}) \oplus \overline{V[X]} && \text{by Eqn. (4)} \\
&\quad = \delta V[X] \oplus (g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \dot{-} \delta V[X]) \oplus \overline{V[X]} && \text{by Eqn. (3)} \\
&\quad = g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \oplus \delta V[X] \oplus \overline{V[X]} && \text{by Eqn. (3)} \\
&\quad = g(V[X_1], ..., \mathit{delta}, ..., V[X_k]) \oplus V[X] && \text{by the ind. hyp.} \\
&\quad = V[X]'
\end{aligned}$$

□

The next lemma characterizes the value for $V[X]$ on each iteration, in terms of certain values that arose in the past. (This will be used in the proof of Theorem 25 to establish that Alg. 8 reaches a fixed point.)

Lemma 24 *For all $X \in NT$, each time that control reaches line 3 of Alg. 8,*

$$V[X] \sqsubseteq \bigoplus_{X \to g(X_1,...,X_k) \in \mathit{Prods}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]}).$$

PROOF. The proof is by induction on the number of times that control reaches line 3.

Base case: The property is established by lines 1 and 2.

Induction step: Assume that the property holds the i^{th} time that control reaches line 3. Let $V[Y]^s$ be the value of $V[Y]$ for the Y that will be selected at line 4. By Lemma 23, $V[Y]^s = V[Y]^o \oplus \mathit{delta}$, where $V[Y]^o$ and delta are the values of $\overline{V[Y]}$ and $\delta V[Y]$, respectively, just before line 4.

By the induction hypothesis, at line 4 we have, for all $X \in \mathrm{NT}$,

$$\begin{aligned}
V[X] \sqsubseteq & \bigoplus_{X \to g(X_1,\ldots,X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]}) \\
= & \bigoplus_{X \to g(X_1,\ldots,Y,\ldots,X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, V[Y]^o, \ldots, \overline{V[X_k]}) \\
& \oplus \bigoplus_{X \to g(X_1,\ldots,X_k) \in \mathrm{Prods}, Y \notin \mathrm{RHS}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]})
\end{aligned}$$

After the loop on lines 6–8 executes, we have

$$\begin{aligned}
V[X] \sqsubseteq & \bigoplus_{X \to g(X_1,\ldots,Y,\ldots,X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, V[Y]^o, \ldots, \overline{V[X_k]}) \\
& \oplus \bigoplus_{X \to g(X_1,\ldots,X_k) \in \mathrm{Prods}, Y \notin \mathrm{RHS}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]}) \\
& \oplus \bigoplus_{X \to g(X_1,\ldots,Y,\ldots,X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, delta, \ldots, \overline{V[X_k]}) \\
= & \bigoplus_{X \to g(X_1,\ldots,Y,\ldots,X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, V[Y]^o \oplus delta, \ldots, \overline{V[X_k]}) \\
& \oplus \bigoplus_{X \to g(X_1,\ldots,X_k) \in \mathrm{Prods}, Y \notin \mathrm{RHS}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]}) \\
= & \bigoplus_{X \to g(X_1,\ldots,Y,\ldots,X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, V[Y]^s, \ldots, \overline{V[X_k]}) \,, \\
& \oplus \bigoplus_{X \to g(X_1,\ldots,X_k) \in \mathrm{Prods}, Y \notin \mathrm{RHS}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]})
\end{aligned}$$

which reestablishes the property for the next time that control reaches line 3: $V[Y]^s$ is the value that $V[Y]$ had the last time that Y was selected at line 4—i.e., $V[Y]^s = \overline{V[Y]}$.

□

The presence of "$\sqsubseteq$" in Lemma 24, rather than "=", is what requires us to phrase the argument in the next theorem in terms of the maximum fixed point of H, rather than in terms of the maximum fixed point of G.

Theorem 25 *Alg. 8 finds the maximum fixed-point solution (and hence the meet-over-all-derivations solution) of a given grammar-flow-analysis problem.*

PROOF. Let $\overrightarrow{W_0} \sqsupseteq \overrightarrow{W_1} \sqsupseteq \ldots \sqsupseteq \overrightarrow{W_r}$ be the sequence of values for $\overrightarrow{V}$ at line 3 on any run of Alg. 8.

First, we must show that Alg. 8 always terminates. We want to argue that each iteration of the loop on lines 3–8 makes a certain amount of progress. In particular, we would like to show that each time a particular nonterminal Z is selected on line 4, it has a strictly smaller value than the previous time that Z was selected; because there are a finite number of nonterminals and $(S, \oplus)$ has no infinite descending chains, this implies that Alg. 8 must terminate.

For Z to be selected on line 4, $\delta V[Z] \sqsubset 0$ must hold. However, $\delta V[Z]$ can only receive a non-0 value in line 7—and only when the value of subexpression $(g(V[Z_1], \ldots, delta, \ldots, V[Z_k]) \mathbin{\dot{-}} V[Z])$ is non-0. Moreover, whenever this happens, it cannot be the case that $V[Z] \sqsubseteq g(V[Z_1], \ldots, delta, \ldots, V[Z_k])$, or else, by Eqn. (5), $(g(V[Z_1], \ldots, delta, \ldots, V[Z_k]) \mathbin{\dot{-}} V[Z])$ would be 0. This, in turn, implies that the assignment on line 8 must also cause the value of $V[Z]$ to decrease. In other words, each decrease in the value of $\delta V[Z]$—which must occur for Z to be selected on line 4—must be accompanied by a decrease in the value of $V[Z]$. Because $(S, \oplus)$ has no infinite descending chains, $V[Z]$ can decrease in value only a finite number of times; consequently, each Z can be selected only a finite number of times, and Alg. 8 must terminate.

Second, we show that $\overrightarrow{W_r}$ is a fixed point of H. When Alg. 8 terminates, for all $X \in \mathrm{NT}$, $W_r[X] = \overline{V[X]}$. Consequently,

$$\begin{aligned}
(H(\overrightarrow{W_r}))[X] &= (\overrightarrow{W_r} \oplus G(\overrightarrow{W_r}))[X] \\
&= \overline{V[X]} \oplus \bigoplus_{X \to g(X_1, \ldots, X_k) \in \mathrm{Prods}} g(\overline{V[X_1]}, \ldots, \overline{V[X_k]}) \\
&= \overline{V[X]} && \text{by Lemma 24} \\
&= W_r[X]
\end{aligned}$$

which means that $\overrightarrow{W_r}$ is a fixed point of H.

Third, we show by induction that, for $0 \leq i \leq r$, $\overrightarrow{W_i} \sqsupseteq \overrightarrow{m_G}$:

Base case: Let $\overrightarrow{V_1} = H(\overrightarrow{0})$ be the value in hand after the first round of non-differential propagation. Because the first round of non-differential propagation can only produce non-0 values for productions with ε on the right-hand side, $\overrightarrow{W_0} = \overrightarrow{V_1}$. Moreover, $\overrightarrow{V_1} \sqsupseteq \overrightarrow{m_G}$ by Theorem 13.

Induction step: Assume that $\overrightarrow{W_i} \sqsupseteq \overrightarrow{m_G}$. By Lemma 21, for all $X \in \mathrm{NT}$, $\delta V[X] \sqsupseteq V[X]$; in particular, for whatever Y is selected in line 4, $\delta V[Y] \sqsupseteq V[Y]$. Consequently, because every production function is monotonic in each argument, each time that control reaches line 8 we have

$$g(V[X_1], \ldots, \delta V[Y], \ldots, V[X_k]) \sqsupseteq g(V[X_1], \ldots, V[Y], \ldots, V[X_k]),$$

which implies that one application of H causes values to decrease at least as much as one application of line 8.

Let n_i be the number of iterations of the loop on lines 6–8 that are performed during the i^{th} iteration of the while loop. Thus, $\overrightarrow{W_{i+1}} \sqsupseteq H^{n_i}(\overrightarrow{W_i})$.

That is, n_i rounds of non-differential propagation via H cause values to decrease at least as much as one iteration of the while loop during differential propagation.

Finally, we observe that

$$\begin{aligned}
\overrightarrow{W_{i+1}} &\sqsupseteq H^{n_i}(\overrightarrow{W_i}) \\
&\sqsupseteq H^{n_i}(\overrightarrow{m_G}) && \text{by the induction hypothesis} \\
&= \overrightarrow{m_G} && \text{by Theorem 13}
\end{aligned}$$

The last step follows from the fact that $\overrightarrow{m_G}$ is the meet-over-all-derivations solution, which by Theorem 13 equals the maximum fixed point of G (and hence of H).

The argument given above shows that, for $0 \leq i \leq r$, $\overrightarrow{W_i} \sqsupseteq \overrightarrow{m_G}$. In particular, for the final value $\overrightarrow{W_r}$, we have $\overrightarrow{W_r} \sqsupseteq \overrightarrow{m_G}$. By Theorem 13, $\overrightarrow{m_G}$ equals the maximum fixed point of H, but in the second part of the proof we showed that $\overrightarrow{W_r}$ is a fixed point of H. Taken together, these facts imply that $\overrightarrow{W_r} = \overrightarrow{m_G}$. $\square$