Interprocedural Dataflow Analysis over Weight Domains with Infinite Descending Chains*

Morten Kühnrich[2], Stefan Schwoon[1], Jiří Srba[2], and Stefan Kiefer[1]

[1] Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{kiefer,schwoon}@in.tum.de
[2] Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark
{mokyhn,srba}@cs.aau.dk

Abstract. We study generalized fixed-point equations over idempotent semirings and provide an efficient algorithm for the detection whether a sequence of Kleene's iterations stabilizes after a finite number of steps. Previously known approaches considered only bounded semirings where there are no infinite descending chains. The main novelty of our work is that we deal with semirings without the boundedness restriction. Our study is motivated by several applications from interprocedural dataflow analysis. We demonstrate how the reachability problem for weighted pushdown automata can be reduced to solving equations in the framework mentioned above and we describe a few applications to demonstrate its usability.

1 Introduction

Weighted pushdown systems [19] are a suitable model for analyzing programs with procedures. They have been used successfully in a number of applications, e.g. BDD-based model checking [22, 7], trust-management systems [10], path optimization [13], and interprocedural dataflow analysis (see [18] for a survey).

The main idea is that the transitions of a pushdown system are labelled with values from a given data domain (e.g. natural numbers). These values can be composed when executed in sequence (e.g. using the addition on natural numbers) and one is then interested in a number of verification questions like reachability of a given configuration with the combined value over all paths leading into this configuration (e.g. by taking the minimum value over all such paths). It has been shown that there are efficient polynomial time algorithms for answering these questions [19].

In this paper, we contribute to the research in this area. We first draw a connection between reachability in weighted pushdown systems (WPDS) over an

* The second and fourth authors are supported in part by the DFG project *Algorithms for Software Model Checking*. The third author is supported in part by Institute for Theoretical Computer Science, project No. 1M0545.

idempotent semiring and solving fixed-point equations over the same semiring. Unlike related work, we allow for infinite descending chains in our semirings (our approach e.g. includes the integer semiring). Due to this reason, the system of equations constructed from a WPDS may not have a solution. We therefore provide an efficient algorithm that either determines the solution or detects the presence of an infinite descending chain. In the latter case, we output some component (variable) of the system affected by the problem. So on one hand we treat domains with infinite descending chains but on the other hand, two restrictions are necessary to make this possible. However, as argued in Section 4, the framework still includes a number of interesting applications.

For better readability some proofs have been moved to an appendix.

1.1 Dataflow Analysis and Fixed-Point Equations

Static analysis gathers information about a program without executing it. Dataflow analysis is an instance of static analysis: it reasons about run-time values of variables or expressions. More to the point, we desire to establish facts that hold at some control point whenever an execution reaches it.

Most approaches to dataflow analysis reduce the problem (explicitly or implicitly) to solving a system of fixed-point equations over some algebraic structure, e.g. a lattice or a semiring. They map the control-flow graph of a program to an equation system $\boldsymbol{X} = \boldsymbol{f}(\boldsymbol{X})$, where the vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ stands for the nodes in the control flow graph, and takes values from some dataflow domain. The vector $\boldsymbol{f} = (f_1, \ldots, f_n)$ stands for the edges in the graph, i.e., the *transfer function* $f_i(\boldsymbol{X})$ describes the effect of the program on X_i in terms of the other dataflow values. Under certain conditions (e.g., the functions f_i are distributive) the desired dataflow information is precisely the greatest solution of the system $\boldsymbol{X} = \boldsymbol{f}(\boldsymbol{X})$, i.e., the greatest fixed point $gfp(\boldsymbol{f})$ of $\boldsymbol{f}$ [17, 21].

There is a large body of literature dealing with dataflow analysis along these lines. Of particular interest to us are interprocedural analyses. The seminal work of Sharir and Pnueli [21] shows how to set up an equation system that captures only the *interprocedurally valid* paths, i.e. those paths in which all return statements lead back to the site of the most recent call. However, [21] computes only one dataflow value for each program point, merging together all the paths that reach it, regardless of the calling context. In [19] a generalization was provided, where the solution of the equations computes a solution for each *configuration*, where configuration denotes a program point together with its calling context. Thus, [19] allows to distinguish dataflow values for different, arbitrary calling contexts. (The merged information can still be obtained as a special case.) The results of [19] were phrased in terms of weighted pushdown systems (WPDS), and we will adopt this notion in our paper.

If the dataflow domain satisfies the so-called *descending chain condition* (i.e. each infinite descending chain eventually becomes stationary), $gfp(\boldsymbol{f})$ can be obtained by *Kleene's iteration*: Let $\overline{0}$ be the greatest domain element, and $\overline{\boldsymbol{0}} = (\overline{0}, \ldots, \overline{0})$. Then Kleene's fixed-point theorem guarantees that the sequence

$\overline{\mathbf{0}}, \boldsymbol{f}(\overline{\mathbf{0}}), \boldsymbol{f}(\boldsymbol{f}(\overline{\mathbf{0}})), \ldots$ reaches $gfp(\boldsymbol{f})$ after finitely many steps. Both [21] and [19] require the descending chain condition.

However, the descending chain condition does not always hold. For example, the lattice of non-positive integers with $\sqcap = \min$ and $\sqcup = \max$ does not satisfy the condition because of the infinite descending chain $0, -1, -2, \ldots$. In fact, this chain arises when doing Kleene's iteration on the equation $X = f(X)$ where $f(X) = \min(X, X - 1)$. More to the point, Kleene's iteration on f would fail to terminate. We will show how to overcome this problem.

Previous work (e.g., [19]) has shown that many important analysis problems can be phrased as equation systems, where $\boldsymbol{f}(\boldsymbol{X})$ contains polynomials over *idempotent semirings*. By polynomial, we mean an expression that is built up from variables, constant elements, and the semiring operations '$\oplus$' (combine) and '$\otimes$' (extend).

Recently, fixed-point equations over idempotent semirings have been studied intensively. While the classical solution is to use Kleene's iteration or chaotic iteration, recent work has proposed faster algorithms and better convergence results based on Newton's method [9, 5, 4, 6]. In these works, the boundedness condition is dropped, but replaced by another condition called ω-continuity, requiring that the infimum of every infinite set exists, thus ensuring that a greatest fixed point can always be found. Our work does not require this condition, and a greatest fixed point is not always guaranteed to exist (but our algorithm detects such a case and reports it). The penalty for this is that a different kind of restriction has to be introduced: we require that semirings are totally ordered and that "extend preserves inequality", i.e., $a \otimes c \neq b \otimes c$ for $a \neq b$ and $a, b, c \neq \overline{\mathbf{0}}$.

Our algorithm executes Kleene's iteration, and if the iteration terminates, it outputs the greatest fixed point. If Kleene's iteration fails to terminate, our algorithm will detect this and still terminate, indicating a responsible variable (a so-called *witness component*).

The work closest to ours is the one by Gawlitza and Seidl [8], who consider systems of equations over the integer semiring. Our algorithm can be seen as a generalization of one of their algorithms to totally ordered semirings where extend preserves inequality and to equations over arbitrary polynomials. Moreover, we provide a direct and self-contained proof of the result. Another related work is by Leroux and Sutre [14]. They present an algorithm for computing least fixed-points for monotone *bounded-increasing* functions over integers. On one hand they consider more general functions like e.g. factorials, on the other hand the minimum and maximum functions are not bounded-increasing according to their definition. As a result, their algorithm is not applicable in our setting of weighted pushdown systems.

We proceed as follows: In Section 2, we provide a new algorithm for solving fixed-point equations. Using this result, we design a new algorithm for interprocedural dataflow analysis in Section 3, which is based on WPDS [19] and still requires a polynomial number of semiring operations. Like previous work on WPDS, the algorithm allows to compute dataflow information for each configuration (if desired). Due to the properties of the systems we handle, our algorithm

either returns a solution (if it exists) or reports that none exists (usually indicating an error in the program). We provide several applications of our theory in Section 4.

2 Fixed-Point Equations over Idempotent Semirings

In this section we shall study fixed-point equations over idempotent semirings and Kleene's iterations over vectors of polynomials.

Definition 1 (Idempotent Semiring). *An* idempotent semiring *is a 5-tuple* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *where* D *is a set called the* domain*,* $\overline{0}, \overline{1} \in D$*, and the binary operators* combine *'*$\oplus$*' and* extend *'*$\otimes$*' on* D *satisfy:*

1. $(D, \oplus)$ *is a commutative monoid with* $\overline{0}$ *as its neutral element and* $(D, \otimes)$ *is a monoid with* $\overline{1}$ *as its neutral element,*
2. *extend distributes over combine, i.e.,* $\forall a, b, c \in D : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$*,*
3. $\overline{0}$ *is an annihilator for extend, i.e.,* $\forall a \in D : a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$*, and*
4. *every* $a \in D$ *is idempotent w.r.t. combine, i.e.,* $\forall a \in D : a \oplus a = a$*.*

Definition 2 (Ordering). *We write* $a \sqsubseteq b$ *for* $a, b \in D$ *whenever* $a \oplus b = a$*.*

As we are mainly interested in algorithmic verification approaches, we shall implicitly consider only *computable* semirings where the elements from the domain are effectively representable, operations combine and extend are algorithmically computable and the test on equality is decidable. We will use the big-O-notation for complexity upper-bounds, though it should be always interpreted relative to the complexity of the semiring operations. In the semirings considered in our applications, we can assume that all operations can be performed in $O(1)$ time. Hence the big-O-notation for the semirings mentioned in this paper corresponds to the standard asymptotic complexity.

Lemma 1. *(i) For all* $a, b \in D$ *it holds that* $a \oplus b \sqsubseteq a$*. (ii) For all* $a, b, c \in D$ *it holds that if* $a \sqsubseteq b$ *then* $a \otimes c \sqsubseteq b \otimes c$*.*

The proof of Lemma 1 is straightforward. We shall now define an additional condition on the extend operator that will be used later on in this section.

Definition 3 (Extend Preserves Inequality). *Given an idempotent semiring we say that* extend preserves inequality *if* $a \neq b$ *implies that* $a \otimes c \neq b \otimes c$ *for any* $a, b, c \in D \setminus \{\overline{0}\}$*.*

Example 1. The tuple $\mathcal{S}_{int} = (\mathbb{Z}_\infty, \min, +, \infty, 0)$ is an idempotent semiring. The domain are the integers extended with infinity $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ where $\min(\infty, a) = \min(a, \infty) = a$ and $a + \infty = \infty + a = \infty$ for all $a \in \mathbb{Z}_\infty$. Combine is the minimum and extend is the usual addition on integers. It is easy to see that $\mathcal{S}_{int}$ meets the requirements of Definition 1. It moreover preserves inequality because the addition does so, and $\sqsubseteq$ is a total order.

Another example of an idempotent semirings is $\mathcal{S}_{rat} = (\mathbb{Q}[0,1], \max, *, 0, 1)$ which is the semiring defined over the rationals in the interval from 0 to 1. Here combine is the maximum and extend is the multiplication on rationals. This semiring $\mathcal{S}_{rat}$ also meets the requirements of Definition 1, extend preserves inequality and $\sqsubseteq$ is a total order. □

In what follows we fix an idempotent semiring $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$. We often omit the $\otimes$ sign in "products", i.e., we write ab for $a \otimes b$. We also fix a set $\mathcal{X} = \{X_1, \ldots, X_n\}$ of variables. Now we define vectors of polynomials over $\mathcal{S}$ and their fixed points following [4].

Let $V = D^n$ denote the set of *vectors* over $\mathcal{S}$. We use bold letters to denote vectors, e.g., $\boldsymbol{v} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$. We also write $\boldsymbol{X} = (X_1, \ldots, X_n)$ to arrange the variables from $\mathcal{X}$ in a vector. We extend $\sqsubseteq$ to vectors by setting $\boldsymbol{u} \sqsubseteq \boldsymbol{v}$ if $\boldsymbol{u}_i \sqsubseteq \boldsymbol{v}_i$ for all $1 \leq i \leq n$.

A *monomial* is a finite expression $a_1 X_{i_1} a_2 X_{i_2} \cdots a_s X_{i_s} a_{s+1}$ where $s \geq 0$, $a_1, \ldots, a_{s+1} \in D$ and $X_{i_1}, \ldots, X_{i_s} \in \mathcal{X}$. A *polynomial* is an expression of the form $m_1 \oplus \cdots \oplus m_s$ where $s \geq 0$ and $m_1, \ldots, m_s$ are monomials. The value of a monomial $m = a_1 X_{i_1} a_2 \cdots a_s X_{i_s} a_{s+1}$ at $\boldsymbol{v}$ is $m(\boldsymbol{v}) = a_1 \boldsymbol{v}_{i_1} a_2 \cdots a_s \boldsymbol{v}_{i_s} a_{s+1} \in D$. The value of a polynomial $f = m_1 \oplus \cdots \oplus m_s$ at $\boldsymbol{v}$ is $f(\boldsymbol{v}) = m_1(\boldsymbol{v}) \oplus \cdots \oplus m_s(\boldsymbol{v})$. A polynomial induces a mapping from V to D that assigns to $\boldsymbol{v}$ the element $f(\boldsymbol{v})$. A vector of polynomials $\boldsymbol{f} = (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n)$ is an n-tuple of polynomials; it induces a mapping from V to V that assigns to a vector $\boldsymbol{v}$ the vector $\boldsymbol{f}(\boldsymbol{v}) = (\boldsymbol{f}_1(\boldsymbol{v}), \ldots, \boldsymbol{f}_n(\boldsymbol{v}))$. A *fixed point of* $\boldsymbol{f}$ is a vector $\boldsymbol{v}$ that satisfies $\boldsymbol{v} = \boldsymbol{f}(\boldsymbol{v})$. A greatest fixed point of $\boldsymbol{f}$ is a fixed point $\boldsymbol{v}$ such that $\boldsymbol{v}' \sqsubseteq \boldsymbol{v}$ holds for all other fixed points $\boldsymbol{v}'$. The size $K(\boldsymbol{f})$ of a vector of polynomials $\boldsymbol{f}$ is the total number of $\oplus$ and $\otimes$ operators in $\boldsymbol{f}$. In particular, given a vector $\boldsymbol{v}$, it takes $O(K(\boldsymbol{f}))$ time to compute $\boldsymbol{f}(\boldsymbol{v})$.

Example 2. Consider the semiring $\mathcal{S}_{int}$ from Example 1. Let $\mathcal{X} = \{X_1, X_2, X_3\}$. Then $\boldsymbol{f} = (-2 \oplus X_2 \otimes X_3,\ X_3 \otimes 1,\ X_1 \oplus X_2)$ is a vector of polynomials over $\mathcal{S}_{int}$. It can be rewritten as $\boldsymbol{f} = (\min\{-2, X_2 + X_3\},\ X_3 + 1,\ \min\{X_1, X_2\})$. The size $K(\boldsymbol{f})$ equals 4. □

It is easy to see that polynomials are monotone and continuous mappings w.r.t. $\sqsubseteq$, see Lemma 1. Kleene's theorem can then be applied (see e.g. [12]), which leads to the following proposition.

Proposition 1. *Let* $\boldsymbol{f}$ *be a vector of polynomials. Let the* Kleene sequence $(\boldsymbol{\kappa}^{(k)})_{k \in \mathbb{N}}$ *be defined by* $\boldsymbol{\kappa}^{(0)} = \overline{\boldsymbol{0}}$ *and* $\boldsymbol{\kappa}^{(k+1)} = \boldsymbol{f}(\boldsymbol{\kappa}^{(k)})$.

(a) We have $\boldsymbol{\kappa}^{(k+1)} \sqsubseteq \boldsymbol{\kappa}^{(k)}$ *for all* $k \in \mathbb{N}$.
(b) If a greatest fixed point exists then it is the infimum of $\{\boldsymbol{\kappa}^{(k)} \mid k \in \mathbb{N}\}$.
(c) If the infimum of $\{\boldsymbol{\kappa}^{(k)} \mid k \in \mathbb{N}\}$ *exists then it is the greatest fixed point.*

Proposition 1 is the mathematical basis for the classical fixed-point iteration: apply $\boldsymbol{f}$ until a fixed point is reached, which is, by Proposition 1 (c), the greatest fixed point of $\boldsymbol{f}$. We call this method *Kleene's iteration*. In general, Kleene's iteration does not always reach a fixed point. Some equations, like $X = X \otimes (-1)$

over $\mathcal{S}_{int}$, do not have any (greatest) fixed point, other equations might have a greatest fixed point but it is not achievable in a finite number of Kleene's iterations (consider for example the above equation but over the semiring $\mathcal{S}_{int}$ extended with the element $-\infty$). It is not a priori clear how to detect whether Kleene's iteration terminates, i.e., computes the greatest fixed point in a finite number of iterations.

Algorithm 1 (called "safe Kleene's iteration") solves this problem. If Kleene's iteration reaches the greatest fixed point, then the algorithm computes it. Otherwise it outputs a *witness component* where Kleene's iteration does not terminate. Formally, a witness component is defined as follows.

Definition 4 (Witness Component). *Let $\boldsymbol{f}$ be a vector of polynomials over an idempotent semiring. A component i $(1 \leq i \leq n)$ is a* witness component *if $\{\boldsymbol{\kappa}_i^{(k)} \mid k \geq 0\}$ is an infinite set.*

In our applications, the presence of a witness component pinpoints a problem of the analyzed model which the user may want to fix. More details are given in Section 4.

Algorithm 1 is based on the generalized Bellman-Ford algorithm of [8] for $\mathcal{S}_{int}$ and generalizes it further to totally ordered semirings where extend preserves inequality and to equations over arbitrary polynomials.

Algorithm 1 Safe Kleene's iteration

Input: A vector of polynomials $\boldsymbol{f} = (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n)$ over an idempotent semiring $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ s.t. $\sqsubseteq$ is a total order and where extend preserves inequality.
Output: Greatest fixed point of $\boldsymbol{f}$ or a witness component.
1: $\boldsymbol{\kappa}^{(0)} := \overline{\mathbf{0}}$
2: **for** $k := 1$ **to** $n + 1$ **do**
3: $\quad \boldsymbol{\kappa}^{(k)} := \boldsymbol{f}(\boldsymbol{\kappa}^{(k-1)})$
4: **end for**
5: **if** $\exists i$ with $1 \leq i \leq n$ such that $\boldsymbol{\kappa}_i^{(n+1)} \neq \boldsymbol{\kappa}_i^{(n)}$ **then**
6: $\quad$ **return** "Kleene's iteration does not terminate. Component i is a witness."
7: **else**
8: $\quad$ **return** "The vector $\boldsymbol{\kappa}^{(n)}$ is the greatest fixed point."
9: **end if**

Theorem 1. *Algorithm 1 is correct and terminates in time $O(n \cdot K(\boldsymbol{f}))$.*

Algorithm 1 on its own is very straightforward, and its proof for polynomials of degree only 1 would directly mimic the proof of Bellman-Ford algorithm. Our contribution is that we prove that it works also for polynomials of higher degrees where more involved technical treatment is necessary. Full details can be found in Appendix A.

Remark 1. In the integer semiring $\mathcal{S}_{int}$, Algorithm 1 can be extended such that it computes *all* witness components and for the remaining terminating components

returns the exact value. This is done as follows. The main loop on lines 2–4 is run once again, but the components that still change are assigned a new semiring element "$-\infty$" on which the operators "+" and "min" act as expected. Thus, $-\infty$ may be propagated through the components during the repetition of the main loop. At the end, all components that are not $-\infty$ have reached their final value, all others can be reported as witness components. For details see [8].

Example 3. Consider again the vector of polynomials from Example 2:

$$\boldsymbol{f} = (\min\{-2, X_2 + X_3\},\ X_3 + 1,\ \min\{X_1, X_2\})\ .$$

Kleene's iteration produces the following Kleene sequence: $\boldsymbol{\kappa}^{(0)} = (\infty, \infty, \infty)$, $\boldsymbol{\kappa}^{(1)} = (-2, \infty, \infty)$, $\boldsymbol{\kappa}^{(2)} = (-2, \infty, -2)$, $\boldsymbol{\kappa}^{(3)} = (-2, -1, -2)$, $\boldsymbol{\kappa}^{(4)} = (-3, -1, -2)$. As $\boldsymbol{\kappa}_1^{(3)} = -2 \neq -3 = \boldsymbol{\kappa}_1^{(4)}$, Alg. 1 returns the first component as a witness. □

Notice that Algorithm 1 merely indicates whether a greatest fixed point can be found *using Kleene's iteration* or not. Even if Algorithm 1 outputs a witness component, a greatest fixed point may still exist (and be found by other means). An example is a semiring over the reals which can admit the sequence $1/2^n$ for some variable. This sequence converges to 0, but Kleene's iteration fails to detect this. Nevertheless, for some semirings like $\mathcal{S}_{int}$ used in our applications, we can make the following stronger statement.

Corollary 1. *Algorithm 1 applied to polynomials over the semiring $\mathcal{S}_{int}$ finds the greatest fixed point iff it exists. If it does not exist, all witness components can be explicitly marked.*

Proof. In $\mathcal{S}_{int}$ a component is a witness component iff Kleene's iteration does not terminate in that component. The rest follows from Definition 4, Proposition 1 and Remark 1. □

3 Weighted Pushdown Systems

In this section we will use the fixed-point equations studied in the previous section for reasoning about properties of weighted pushdown systems (WPDS) [19]. We are interested in applying Theorem 1 to weighted pushdown systems; therefore we implicitly consider only semirings that are totally ordered, and where extend preserves inequality.

Definition 5 (Weighted Pushdown System). *A weighted pushdown system is a 4-tuple $\mathcal{W} = (P, \Gamma, \Delta, \mathcal{S})$, where P is a finite set of* control states*, Γ is a finite* stack alphabet*, $\Delta \subseteq (P \times \Gamma) \times D \times (P \times \Gamma^*)$ is a finite set of* rules*, and $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is an idempotent semiring.*

We write $pX \overset{d}{\hookrightarrow} q\alpha$ whenever $r = (p, X, d, q, \alpha) \in \Delta$ and call d the *weight* of r, denoted by d_r. We consider only rules where $|\alpha| \leq 2$. (It is well-known that every WPDS can be translated into a one that obeys this restriction and is larger by only a constant factor, see, e.g., [20]. The reduction preserves reachability.) We let the symbols X, Y, Z range over Γ and α, β, γ range over Γ^*.

Example 4. As a running example in this section, we consider a weighted pushdown system over the semiring with both positive and negative integers as weights, i.e. $\mathcal{W}_{ex} = (\{p, q\}, \{X, Y\}, \Delta_{ex}, \mathcal{S}_{int})$, where $\Delta_{ex} = \{pX \overset{1}{\hookrightarrow} qY,\ \ pX \overset{1}{\hookrightarrow} pXY,\ \ pY \overset{1}{\hookrightarrow} p,\ \ qY \overset{-2}{\hookrightarrow} q\}$. □

A *configuration* of a weighted pushdown system $\mathcal{W}$ is a pair $p\gamma$ where $p \in P$ and $\gamma \in \Gamma^*$. A transition relation $\Rightarrow$ on configurations is defined by $pX\gamma \overset{r}{\Rightarrow} q\alpha\gamma$ iff $\gamma \in \Gamma^*$ and there exists $r \in \Delta$, where $r = (pX \overset{d}{\hookrightarrow} q\alpha)$. We annotate $\Rightarrow$ with the rule $r \in \Delta$ which was used to derive the conclusion. If there exists a sequence of configurations $c_0, \ldots, c_n$ and rules $r_1, \ldots, r_n$ such that $c_{i-1} \overset{r_i}{\Rightarrow} c_i$ for all $i = 1, \ldots, n$, then we write $c_0 \overset{\sigma}{\Rightarrow} c_n$, where $\sigma := r_1 \ldots r_n$. The weight of σ is defined as $v(\sigma) = d_{r_1} \otimes \cdots \otimes d_{r_n}$. By definition $v(\epsilon) = \overline{1}$.

Let c, c' be two configurations and $\sigma \in \Delta^*$ such that $c \overset{\sigma}{\Rightarrow} c'$. We call c a *predecessor* of c' and c' a *successor* of c. In the following, we will consider the problem of computing the set of all predecessors $pre^*(c_f)$ and successors $post^*(c_f)$ for a given configuration c_f. Due to space limitations we provide the full treatment only for the predecessors; the computation of successors is analogous and it is provided in Appendix C.

Let us fix a WPDS $\mathcal{W}$ and a *target configuration* c_f, where $c_f = p_f\epsilon$ for some control state p_f. For any configuration c of $\mathcal{W}$, we want to know the minimal weight of a path from c to c_f. If a path of minimal weight does not exist for every c, we want to detect such a case. In our applications (see Section 4), this situation usually indicates the existence of an error.

Remark 2. In the literature, it is more common to consider a *regular* set C of target configurations. This problem, however, reduces to the one with only a single target configuration c_f. The reduction can be achieved by extending $\mathcal{W}$ with additional 'pop' rules that simulate a finite automaton for C; the 'pop' rules will succeed in reducing the stack to c_f iff they begin with a configuration in C. For details, see [19], Section 3.1.1.

At an abstract level, we are interested in solutions for the following equation system, in which each configuration c is represented by a variable $[c]$. Intuitively, the greatest solution (if it exists) for the variable $[c]$ will correspond to the minimum (w.r.t. the combine operator) of accumulated weights over all paths leading from the configuration c to c_f.

$$[c] = I(c) \oplus \bigoplus_{c \overset{r}{\Rightarrow} c'} (d_r \otimes [c']), \qquad \text{where } I(c) := \begin{cases} \overline{1} & \text{if } c = c_f \\ \overline{0} & \text{otherwise} \end{cases} \tag{1}$$

Let us consider the Kleene sequence $(\boldsymbol{\kappa}^{(k)})_{k \in \mathbb{N}}$ for (1). By $\boldsymbol{\kappa}^{(k)}_{[c]}$ we denote the entry for configuration c in the k-th iteration of the Kleene sequence.

Lemma 2. *For $k \geq 1$ and any configuration c, the following holds*

$$\boldsymbol{\kappa}^{(k)}_{[c]} = \bigoplus \{\, v(\sigma) \mid c \overset{\sigma}{\Rightarrow} c_f,\ |\sigma| < k \,\} \ .$$

Thus, $[c]$ is a witness component of (1) iff no path of minimal weight exists, because it is possible to construct longer and longer paths with smaller and smaller weights. On the other hand, if (1) has a greatest fixed point, then the fixed point at $[c]$ gives the combine of the weights of all sequences leading from c to c_f, commonly known as the *meet-over-all-paths*. However, (1) defines an infinite system of equations, which we cannot handle directly. In the following, we shall derive a *finite* system of equations, from which we can determine the greatest fixed point of (1) or the existence of a witness component.

Definition 6 (Pop Sequence). *Let p, q be control states and X be a stack symbol. A* pop sequence *for p, X, q is any sequence $\sigma \in \Delta^*$ such that $pX \stackrel{\sigma}{\Rightarrow} q\epsilon$.*

Let us consider the following polynomial equation system, in which the variables are triples $[pXq]$, where p, q are control states and X a stack symbol:

$$[pXq] = \bigoplus_{(pX \stackrel{d}{\hookrightarrow} q\epsilon) \in \Delta} d \quad \oplus \bigoplus_{(pX \stackrel{d}{\hookrightarrow} rY) \in \Delta} \left(d \otimes [rYq]\right) \oplus \bigoplus_{(pX \stackrel{d}{\hookrightarrow} rYZ) \in \Delta} \left(d \otimes \bigoplus_{s \in P} \left([rYs] \otimes [sZq] \right) \right) . \quad (2)$$

Intuitively, Equation (2) lists all the possible ways in which a pop sequence for p, X, q can be generated and computes the values accumulated along each of them.

Example 5. Let us consider the WPDS $\mathcal{W}_{ex}$ from Example 4. Here, the scheme presented in (2) yields a system with eight variables and equations, four of which are reproduced below.

$$\begin{array}{ll} [pXp] = \min\{1 + [qYp],\ 1 + [pXp] + [pYp],\ 1 + [pXq] + [qYp]\} & [pYp] = 1 \\ [pXq] = \min\{1 + [qYq],\ 1 + [pXp] + [pYq],\ 1 + [pXq] + [qYq]\} & [qYq] = -2 \end{array}$$

Notice that the other four variables would be simply assigned to the $\overline{0}$ element, in this case ∞. □

We now examine the Kleene sequence $(\boldsymbol{\kappa}^{(k)})_{k \in \mathbb{N}}$ for (2).

Lemma 3. *For any $k \geq 1$, control states p, q, and stack symbol X,*

$$\bigoplus \{\, v(\sigma) \mid c \stackrel{\sigma}{\Rightarrow} c_f,\ |\sigma| \leq 2^{k-1} \,\} \sqsubseteq \boldsymbol{\kappa}^{(k)}_{[pXq]} \sqsubseteq \bigoplus \{\, v(\sigma) \mid c \stackrel{\sigma}{\Rightarrow} c_f,\ |\sigma| \leq k-1 \,\} .$$

Thus, $[pXq]$ is a witness component of (2) iff no minimal-weight pop sequence exists for p, X, q. On the other hand, if no witness component exists, then the value of $[pXq]$ in the greatest fixed point denotes the combine of the weights of all pop sequences for p, X, q.

We now show how (2) can be used to derive statements about (1). Let a configuration $c = pX_1 \dots X_n$ be a predecessor of c_f. Then any sequence σ leading from c to c_f can be subdivided into subsequences $\sigma_1, \dots, \sigma_n$ and there exist states $p =: p_0, p_1, \dots, p_{n-1}, p_n := p_f$ such that σ_i is a pop sequence for p_{i-1}, X_i, p_i, for all $i = 1, \dots, n$. As a consequence, we can obtain a solution for (1) from a solution

for (2): suppose that $\boldsymbol{\lambda}$ is the greatest fixed point of (2), and let $\boldsymbol{\mu}$ be a vector of configurations as follows:

$$\boldsymbol{\mu}_{[c]} = \bigoplus_{p_1,\ldots,p_{n-1}} \left(\boldsymbol{\lambda}_{[pX_1p_1]} \otimes \cdots \otimes \boldsymbol{\lambda}_{[p_{n-1}X_np_f]}\right), \qquad \text{for } c = pX_1 \ldots X_n \ . \quad (3)$$

It is easy to see that (3) "sums up" all possible paths from c to c_f, and therefore yields the meet-over-all-paths for c. Thus, $\boldsymbol{\mu}$ is a solution (greatest fixed point) of (1). On the other hand, if (1) has a witness component, then (2) must also have one.

Theorem 2. *Applying Algorithm 1 to (2) either yields a witness component or, via (3), the greatest fixed point of (1).*

Example 6. Once again, consider $\mathcal{W}_{ex}$ from Example 4 and the equation system from Example 5. Here, the Kleene sequence quickly converges to the values 1 for $[pYp]$, -2 for $[qYq]$, and ∞ for all other variables except $[pXq]$, which turns out to be a witness component of (2). Indeed, one can construct a series of pop sequences for p, X, q with smaller and smaller weights, e.g. $pX \overset{1}{\Rightarrow} qY \overset{-2}{\Rightarrow} q\epsilon$, and $pX \overset{1}{\Rightarrow} pXY \overset{1}{\Rightarrow} qYY \overset{-2}{\Rightarrow} qY \overset{-2}{\Rightarrow} q\epsilon$, and etc. with weights -1, -2 etc. If $c_f = q\epsilon$, this implies that, e.g., pX is a witness component of (1). On the other hand, qY or qYY would not be a witness components, because their values in (3), would not be affected by the variable $[pXq]$ and evaluate to -2 and -4, respectively. □

Remark 3. The size of the equation system (2) is polynomial in $\mathcal{W}$. Notice that it makes sense to generate equations only for such triples p, X, q in which pX occurs on the left-hand side or right-hand side of some rule. Under this assumption, the number of equations in (2) is $\mathcal{O}(|P| \cdot |\Delta|)$, and its overall size is $\mathcal{O}(|P|^2 \cdot |\Delta|)$, the same complexity as in the algorithms for computing predecessors in [3]. According to Theorem 1, Algorithm 1 therefore runs in $\mathcal{O}(|P|^3 \cdot |\Delta|^2)$ time on (2). For any configuration c of interest, the value $\boldsymbol{\mu}_c$ in (3) can be easily obtained from the result of Algorithm 1. See also the $\mathcal{W}$-automaton technique in the subsection to follow. A similar conclusion about the complexity of the algorithm for computing successors can be drawn thanks to the (linear) connection between forward and backward reachability analysis described in Appendix C.

3.1 Weighted Automata

For (unweighted) pushdown systems, it is well-known that reachability preserves regularity; in other words, given a regular set of configurations, the set of all predecessors resp. successors is regular. Moreover, given a finite automaton recognizing a set of configurations, automata recognizing the predecessors or successors can be constructed in polynomial time (see, e.g., [3]).

It is also known that the results carry over to weighted pushdown systems provided that the semiring is *bounded*, i.e., there are no infinite descending chains w.r.t. $\sqsubseteq$ [19]. For this purpose, so-called *weighted automata* are employed.

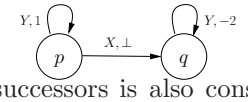
Definition 7 (Weighted $\mathcal{W}$-Automaton). *Let $\mathcal{W} = (P, \Gamma, \Delta, \mathcal{S})$ be a pushdown system over a bounded semiring $\mathcal{S}$. A* $\mathcal{W}$-automaton *is a 5-tuple $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ where Q is a finite set of* states, *$\rightarrow \subseteq Q \times \Gamma \times D \times Q$ is a finite set of* transitions, *$P \subseteq Q$, i.e. the control states of $\mathcal{W}$, are the set of* initial states *and $F \subseteq Q$ is a set of* final (accepting) states.

Let $\pi = t_1 \dots t_n$ be a path in $\mathcal{A}$, where $t_i = (q_i, X_i, d_i, q_{i+1})$ for all $1 \leq i \leq n$. The weight of π is defined as $v(\pi) := d_1 \otimes \dots \otimes d_n$. If $q_1 \in P$ and $q_{n+1} \in F$, then we say that π accepts *the configuration $q_1 X_1 \dots X_n$. Moreover, if c is a configuration, we define $v_{\mathcal{A}}(c)$ as the combine of all $v(\pi)$ such that π accepts c. In this case, we also say that $\mathcal{A}$* accepts *c* with weight *$v_{\mathcal{A}}(c)$.*

In [19] the following problem is considered for the case of bounded semirings: compute a $\mathcal{W}$-automaton $\mathcal{A}$ such that $v_A(c)$ equals the meet-over-all-paths (or equivalently the greatest fixed point of (1), which always exists for bounded semirings) from c to c_f, for every configuration c.

We extend this solution to the case of unbounded semirings, using Theorem 2. We first apply Algorithm 1 to the equation system (2). If the algorithm yields the greatest fixed point, then we construct a $\mathcal{W}$-automaton $\mathcal{A} = (P, \Gamma, \rightarrow, P, \{c_f\})$, with $(p, X, d, q) \in \rightarrow$ for all p, X, q such that d is the value of $[pXq]$ in the greatest fixed point computed by Algorithm 1. Given a configuration c, it is easy to see that $v_{\mathcal{A}}(c)$ yields the same result as in (3).

Example 7. The automaton arising from Example 6 is depicted below where the witness component is marked by $\perp$ and transitions with the value ∞ are omitted completely.



□

The problem of computing successors is also considered in [19], i.e., computing a $\mathcal{W}$-automaton $\mathcal{A}$ where $v_{\mathcal{A}}(c)$ is the meet-over-all-paths from an initial configuration c_0 to c. Using our technique, this result can also be extended to unbounded semirings; Appendix C shows an equation system for this problem, which can be converted into a $\mathcal{W}$-automaton for $post^*(c_0)$ in analogous fashion.

4 Applications

Here we outline some applications of the theory developed in this paper. Unless stated otherwise, we will consider the semiring $\mathcal{S}_{int}$ as described in Example 1. Following Remark 1 and Corollary 1, we assume that all nonterminating components can be detected in this semiring and the corresponding transitions in the $\mathcal{W}$-automaton will be assigned the value $\perp$. The terminating components resp. the corresponding transitions in the $\mathcal{W}$-automaton take the computed value.

Note that the previously known approaches to reachability in weighted pushdown automata are not applicable to any of the below presented cases because they required the semiring to be bounded (no infinite descending chains). Boundedness is, however, not satisfied in any of our applications. Our first two applications are new and we are not aware of any other algorithms that could achieve the

same results. Our third application deals with shape-balancedness of context-free languages, a problem for which an algorithm was recently described in [23].

Memory Allocations in Linux Kernel. Correct memory allocation and deallocation is crucial for the proper functionality of an operating system. In Linux the library `linux/gfp.h` is used for allocation and deallocation of kernel memory pages via the functions `alloc_pages` and `_free_pages` respectively. The functions which are argumented with a number n (also called the *order*) allocate or deallocate 2^n memory pages. Citing [15, page 187]:"You must be careful to free only pages you allocate. Passing the wrong `struct page` or address, or the incorrect *order*, can result in corruption." This means that a basic safety requirement is: never free more pages than what are allocated.

As most questions about real programs are in general undecidable, several techniques have been suggested to provide more tractable models. For example so-called boolean programs [2] have recently been used to provide a suitable abstraction via pushdown systems. Assume a given pushdown system abstraction resulting from the program code. The transitions in the pushdown system are labelled with the programming primitives, among others the ones for allocation and deallocation of memory pages. If a given pushdown transition allocates 2^n memory pages, we assign it the weight 2^n; if it deallocates 2^n pages, we assign it the weight -2^n; in all other cases the weight is set to 0.

Now the pushdown abstraction corrupts the memory iff a configuration is reachable from the given initial configuration pX with negative weight. As shown in Section 3, we can in polynomial time (w.r.t. to the input pushdown system $\mathcal{W}$) construct a $\mathcal{W}$-automaton $\mathcal{A}$ for $post^*(\{pX\})$. For technical convenience, we first replace all occurrences of $\perp$ in $\mathcal{A}$ with $-\infty$. From all initial control-states of $\mathcal{A}$ we now run e.g. the Bellman-Ford shortest path algorithm (which can detect negative cycles and assign the weight to $-\infty$ should there be such) to check whether there is a path going to some accept state with an accumulated negative weight. This is doable in polynomial time. If a negative weight path is found this means that the corresponding configuration is reachable with a negative weight, hence there is a memory corruption (at least in the pushdown abstraction). Otherwise, the system is safe. All together our technique gives a polynomial time algorithm for checking memory corruption with respect to the size of the abstracted pushdown system. Also depending on whether under- or over-approximation is used in the abstraction step, our technique can be used for detecting errors or showing the absence of them, respectively.

Correspondence Assertions. In [24] Woo and Lam analyze protocols using the so-called *correspondences* between protocol points. A correspondence property relates the occurrence of a transition to an earlier occurrence of some other transition. In sequential programs (modelled as pushdown systems) assume that assertions of the form `begin` ℓ and `end` ℓ (where ℓ is a label taken from a finite set of labels) are inserted by the programmer into the code. The program is *safe* if for each `end` ℓ reached at a program point there is a unique corresponding `begin` ℓ at an earlier execution point of the program. Verifying safety via correspondence

assertions can be done using a similar technique as before. For each label ℓ we create a weighted pushdown system based on the initially given boolean program abstraction where every instruction `begin` ℓ has the weight $+1$, every instruction `end` ℓ the weight -1, and all other instructions have the weight 0. Now the pushdown system is safe if and only if every reachable configuration has nonnegative accumulated weight. This can be verified in polynomial time as outlined above.

Shape-Balancedness of Context-Free Languages. In static analysis of programs generating XML strings and in other XML-related questions, the balancedness problem has been recently studied (see e.g. [1, 11, 16]). The problem is, given a context-free language with a paired alphabet of opening and closing tags, to determine whether every word in the language is properly balanced (i.e. whether every opening tag has a corresponding closing tag and vice versa). Tozawa and Minamide recently suggested [23] a polynomial time algorithm for the problem. Their involved algorithm consists of two stages and in the first stage they test for the *shape-balancedness* property, i.e., if all opening tags as well as closing tags are treated as of the same sort, is every accepted word balanced? Assume a given pushdown automaton accepting (by final control-states) the given context-free language. If we label all opening tags with weight $+1$ and all closing tags with weight -1, the shape-balancedness question is equivalent to checking (i) whether every accepted word has the weight equal to 0 and (ii) whether all configurations on every path to some final control-state have nonnegative accumulated weights. Our generic technique provides polynomial time algorithms to answer these questions.

To verify property (i), we first consider the semiring $\mathcal{S}_{int} = (\mathbb{Z}_\infty, \min, +, \infty, 0)$. We now construct in polynomial time for the given initial configuration pX a weighted $post^*(\{pX\})$ $\mathcal{W}$-automaton $\mathcal{A}$, replace all labels $\perp$ with $-\infty$, and for each final control-state q (of the pushdown automaton) we find in $\mathcal{A}$ a shortest path from q to every accept state of $\mathcal{A}$. This can be done in polynomial time using e.g. the Bellman-Ford shortest path algorithm, which can moreover detect negative cycles and set the respective shortest path to $-\infty$. If any of the shortest paths are different from 0, we terminate because the shape-balancedness property is broken. If the system passes the first test, we run the same procedure once more but this time with the semiring $(\mathbb{Z} \cup \{-\infty\}, \max, +, -\infty, 0)$ and where $\perp$ is replaced with ∞, i.e., we are searching for the longest path in the automaton $\mathcal{A}$. Again if at least one of those paths has the accumulated weight different from 0, we terminate with a negative answer. If the pushdown system passes both our tests, this means that any configuration in the set $post^*(\{pX\})$ starting with some final control-state (of the pushdown automaton) is reachable only with the accumulated weight 0 and we can proceed to verify property (ii).

For (ii), we construct the weighted $post^*(\{pX\})$ $\mathcal{W}$-automaton for the integer semiring $\mathcal{S}_{int}$. Now we restrict the automaton to contain only those configurations that can really involve into some accepting configuration by simply intersecting it (by the usual product construction) with the unweighted $\mathcal{W}$-automaton (of polynomial size) representing $pre^*((q_1 + \cdots + q_n)\Gamma^*)$ where $q_1, \ldots, q_n$ are all

final control-states and Γ is the stack alphabet. Property (ii) now reduces to checking whether the product automaton accepts some configuration with negative weight, which can be answered in polynomial time using the technique described in our first application.

Unfortunately, [23] provides no complexity analysis other than the statement that the algorithm is polynomial. Our general-purpose algorithm, on the other hand, immediately provides a precise complexity bound. Consider a given context-free grammar of size n over some paired alphabet. It can be (by the standard textbook construction) translated into a (weighted) pushdown automaton of size $O(n)$ and moreover with a constant number of states. As mentioned in Section 3, this automaton can be normalized in linear time and we can then build a weighted $post^*(\{pX\})$ $\mathcal{W}$-automaton, of size $O(n^2)$ with $O(n)$ states and in time $O(n^4)$. Details can be found in Appendix C. Now running the Bellman-Ford algorithm twice in order to verify property (i) takes only the time $O(n^3)$. In property (ii) the Bellman-Ford algorithm is run on a product of the weighted $post^*$ automaton and an unweighted pre^* automaton, which has only a constant number states. Hence the size of the product is still $O(n^2)$ and Bellman-Ford algorithm will run in time $O(n^3)$ as before. This gives the total running time of $O(n^4)$.

5 Conclusion

We presented a unified framework how to deal with interprocedural dataflow analysis on weighted pushdown automata where the weight domains might contain infinite descending chains. The problem was solved by reformulating it via generalized fixed-point equations which required polynomials of degree two. To the best of our knowledge this is the first approach that enables to handle this kind of domains. On the other hand, we do not consider completely general idempotent semirings as we require that the elements in the domain are totally ordered and that extend preserves inequality. Nevertheless, we showed that our theory is still applicable. Already the reachability analysis of weighted pushdown automata over the integer semiring, one particular instance of our general framework, was not known before and we provided several examples of its potential use in verification.

Regarding the two restrictions we introduced, we claim that the first condition of total ordering can be relaxed to orderings of bounded width, where the maximum number of incomparable elements is bounded by some a priori given constant c. By running the main loop in Algorithm 1 $cn + 1$ times, we should be able to detect nontermination also in this case. The motivation for introducing bounded width comes from the fact that this will allow us to combine (via the product construction) one unbounded domain, like e.g. the integer semiring, with a fixed number of finite domains in order to observe additional properties along the computations. The question whether the second restriction (extend preserves inequality) can be relaxed as well remains open and is a part of our future work.

References

1. J. Berstel and L. Boasson. Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, 2002.
2. A. Bouajjani and J. Esparza. Rewriting models of Boolean programs. In *Proc. RTA*, LNCS 4098, pages 136–150, 2006.
3. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, LNCS 1855, pages 232–247, 2000.
4. J. Esparza, S. Kiefer, and M. Luttenberger. An extension of Newton's method to ω-continuous semirings. In *Proc. DLT*, LNCS 4588, pages 157–168, 2007.
5. J. Esparza, S. Kiefer, and M. Luttenberger. On fixed point equations over commutative semirings. In *STACS'07*, LNCS 4397, pages 296–307. Springer, 2007.
6. J. Esparza, S. Kiefer, and M. Luttenberger. Newton's method for ω-continuous semirings. In *Proc. ICALP, part II*, LNCS 5126, pages 14–26. Springer, 2008.
7. J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *TACAS*, LNCS 3920, pages 489–503, 2006.
8. T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *ESOP'07*, LNCS 4421, pages 300–315. Springer, 2007.
9. M. W. Hopkins and D. Kozen. Parikh's theorem in commutative Kleene algebra. In *Proc. LICS*, pages 394–401. IEEE, 1999.
10. S. Jha, S. Schwoon, H. Wang, and T. Reps. Weighted pushdown systems and trust-management systems. In *Proc. TACAS*, LNCS 3920, pages 1–26, 2006.
11. Ch. Kirkegaard and A. Møller. Static analysis for Java servlets and JSP. In *Proc. SAS*, LNCS 4134, pages 336–352, 2006.
12. W. Kuich. *Handbook of Formal Languages*, volume 1, chapter 9: Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata, pages 609–677. Springer, 1997.
13. A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *Proc. ESOP*, LNCS 3924, pages 246–263, 2006.
14. J. Leroux and G. Sutre. Accelerated data-flow analysis. In *Proc. SAS*, LNCS 4634, pages 184–199, 2007.
15. R. Love. *Linux Kernel Development*. Novell Press, second edition, 2005.
16. Y. Minamide and A. Tozawa. XML validation for context-free grammars. In *Proc. APLAS*, LNCS 4279, pages 357–373, 2006.
17. F. Nielson, H. R. Nielson, and Ch. Hankin. *Principles of Program Analysis*. Springer, 1999.
18. T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *Proc. FSTTCS*, LNCS 4855, pages 23–51, 2007.
19. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58(1–2):206–263, 2005.
20. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, TU Munich, 2002.
21. M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter 7: Two Approaches to Interprocedural Data Flow Analysis, pages 189–233. Prentice-Hall, 1981.
22. D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jMoped: A test environment for Java programs. In *Proc. CAV*, LNCS 4590, pages 164–167, 2007.
23. A. Tozawa and Y. Minamide. Complexity results on balanced context-free languages. In *Proc. FoSSaCS*, LNCS 4423, pages 346–360, 2007.
24. T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proc. SP*, pages 112–118. IEEE, 1993.

Appendix

A Proof of Theorem 1

The statement about the runtime follows immediately from our definition of $K(\boldsymbol{f})$. If Algorithm 1 returns a fixed point, it is the greatest fixed point as it is the result of Kleene's iteration. It remains to show that if the algorithm returns the statement of line 1 then this statement in fact holds.

For that purpose we introduce the concept of derivation trees that was also used in [5, 4]. It generalizes the well-known notion from language theory to semirings. In the following we identify a node x of a tree t with the subtree of t rooted at x. In particular, we identify a tree with its root.

Definition 8 (Derivation Tree). *Let $\boldsymbol{f}$ be a vector of n polynomials. A* derivation tree *t of $\boldsymbol{f}$ is an ordered finite tree whose nodes are labelled with both a variable X_i ($1 \leq i \leq n$) and a monomial m of $\boldsymbol{f}_i$. We write λ_v, resp. λ_m for the corresponding labelling-functions. If $\lambda_m(x) = a_1 X_{i_1} a_2 \dots X_{i_s} a_{s+1}$ for some $s \geq 0$, then x has exactly s children $x_1, \dots, x_s$, ordered from left to right, with $\lambda_v(x_j) = X_{i_j}$ for all $j = 1, \dots, s$.*

Notice that a node x in a derivation tree is a leaf if and only if $\lambda_m(x) = a$ for some constant $a \in D$. The *height* $h(t)$ of a derivation tree t is the length of a longest path from the root to a leaf. For the length, we count the number of nodes on the path including both the root and the leaf. The *yield* $\mathsf{Y}(t)$ of a derivation tree t with $\lambda_m(t) = a_1 X_{i_1} a_2 \cdots X_{i_s} a_{s+1}$ is inductively defined as $\mathsf{Y}(t) = a_1 \mathsf{Y}(t_1) a_2 \cdots \mathsf{Y}(t_s) a_{s+1}$. Figure 1 shows a derivation tree for our running example.

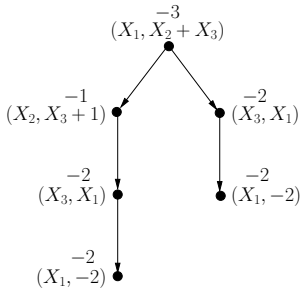


Fig. 1. A derivation tree of height 4 for $\boldsymbol{f} = (\min\{-2, X_2 + X_3\},\ X_3 + 1,\ \min\{X_1, X_2\})$. The labels of a node x are denoted by $(\lambda_v(x), \lambda_m(x))$. The yields are written on top on the labels.

The following proposition is easy to prove by induction on the height (see also [4]).

Proposition 2. *Let $\boldsymbol{f}$ be a vector of n polynomials over a semiring. For all $k \in \{1, 2, \ldots\}$ and all $1 \leq i \leq n$ we have*

$$\boldsymbol{\kappa}_i^{(k)} = \bigoplus \{\, \mathsf{Y}(t) \mid t \text{ is a derivation tree of } \boldsymbol{f} \text{ with } h(t) \leq k \text{ and } \lambda_v(t) = X_i \,\} \ .$$

Notice that the set of yields in Proposition 2 is always finite and may be empty. If it is empty we set $\bigoplus \emptyset = \overline{\mathbf{0}}$. Now we prove the following lemma from which the correctness of Algorithm 1 follows immediately.

Lemma 4. *Let $\boldsymbol{f}$ be a vector of n polynomials over a totally ordered idempotent semiring such that extend preserves inequality. Let $(\boldsymbol{\kappa}^{(i)})_{i \in \mathbb{N}}$ denote its Kleene sequence. If $\boldsymbol{\kappa}_i^{(n)} \neq \boldsymbol{\kappa}_i^{(n+1)}$ for some $1 \leq i \leq n$ then i is a witness component.*

Proof. In this proof we write $a \sqsubset b$ to denote that $a \sqsubseteq b$ and $a \neq b$. We first show the following:

$$\text{If } \boldsymbol{\kappa}_i^{(k)} \sqsubset \boldsymbol{\kappa}_i^{(k-1)} \text{ for some } k > n \text{ then } \boldsymbol{\kappa}_i^{(k')} \sqsubset \boldsymbol{\kappa}_i^{(k)} \text{ for some } k' > k. \tag{4}$$

Let $\boldsymbol{\kappa}_i^{(k)} \sqsubset \boldsymbol{\kappa}_i^{(k-1)}$. By Proposition 2 and using the total order of the semiring, there is a tree t with $\lambda_v(t) = X_i$ such that $\boldsymbol{\kappa}_i^{(k)} = \mathsf{Y}(t)$ and $h(t) = k > n$. So there is a path in t from the root to a leaf and some variable X_j with two nodes x_1, x_2 on the path such that $\lambda_v(x_1) = \lambda_v(x_2) = X_j$. Assume w.l.o.g. that x_1 is closer to the root than x_2. As $\sqsubseteq$ is a total order, one of the following holds.

- If $\mathsf{Y}(x_2) \sqsubseteq \mathsf{Y}(x_1)$ then construct a tree t' from t by replacing the subtree rooted at x_1 by the subtree rooted at x_2. We have $\lambda_v(t') = X_i$ and $h(t') = k'$ for some $k' < k$. By monotonicity of $\otimes$ (Lemma 1 part (ii)) we have $\mathsf{Y}(t') \sqsubseteq \mathsf{Y}(t)$. So $\mathsf{Y}(t) = \boldsymbol{\kappa}_i^{(k)} \overset{\text{Prop. 1(a)}}{\sqsubseteq} \boldsymbol{\kappa}_i^{(k')} \overset{\text{Prop. 2}}{\sqsubseteq} \mathsf{Y}(t') \sqsubseteq \mathsf{Y}(t)$. Hence, $\boldsymbol{\kappa}_i^{(k)} = \boldsymbol{\kappa}_i^{(k')}$ which, by Prop. 1(a), implies $\boldsymbol{\kappa}_i^{(k)} = \boldsymbol{\kappa}_i^{(k-1)}$. This contradicts the assumption that $\boldsymbol{\kappa}_i^{(k-1)} \neq \boldsymbol{\kappa}_i^{(k)}$. So this case does not occur.
- If $\mathsf{Y}(x_1) \sqsubset \mathsf{Y}(x_2)$ then construct a tree t' from t by replacing the subtree rooted at x_2 by the subtree rooted at x_1. We have $\lambda_v(t') = X_i$ and $h(t') = k'$ for some $k' > k$. By monotonicity of $\otimes$ (Lemma 1 part (ii)) and as extend preserves inequality we have $\boldsymbol{\kappa}_i^{(k')} \sqsubseteq \mathsf{Y}(t') \sqsubset \mathsf{Y}(t) = \boldsymbol{\kappa}_i^{(k)}$. So $\boldsymbol{\kappa}_i^{(k')} \sqsubset \boldsymbol{\kappa}_i^{(k)}$.

This proves our claim (4).

It follows from the claim and Proposition 1(a) that if $\boldsymbol{\kappa}_i^{(k)} \sqsubset \boldsymbol{\kappa}_i^{(k-1)}$ for some $k > n$ then $\boldsymbol{\kappa}_i^{(l)} \sqsubset \boldsymbol{\kappa}_i^{(l-1)}$ for some $l > k$. Hence, if $\boldsymbol{\kappa}_i^{(n)} \neq \boldsymbol{\kappa}_i^{(n+1)}$ then $\{\boldsymbol{\kappa}_i^{(k)} \mid k \in \mathbb{N}\}$ is infinite. This completes the proof. □

B Proofs of Lemma 2 and Lemma 3

Lemma 2 claims that in the equation system (1) the following holds for every $k \geq 1$ and any configuration c:

$$\boldsymbol{\kappa}_{[c]}^{(k)} = \bigoplus \{\, v(\sigma) \mid c \overset{\sigma}{\Rightarrow} c_f,\ |\sigma| < k \,\}$$

This follows directly from Proposition 2, and because every derivation tree of height k for (1) corresponds to a sequence of $k-1$ moves in the WPDS. □

Lemma 3 claims that in the equation system (2) the following holds for every $k \geq 1$, control states p, q, and stack symbol X:

$$\bigoplus\{\, v(\sigma) \mid c \stackrel{\sigma}{\Rightarrow} c_f,\ |\sigma| \leq 2^{k-1} \,\} \sqsubseteq \boldsymbol{\kappa}^{(k)}_{[pXq]} \sqsubseteq \bigoplus\{\, v(\sigma) \mid c \stackrel{\sigma}{\Rightarrow} c_f,\ |\sigma| \leq k-1 \,\}$$

A derivation tree of height k for (2) corresponds to a path in $\mathcal{W}$ whose length is at least $k-1$ (if all internal nodes have just one child) and at most 2^{k-1} (if all internal nodes have two children). Because of this, and because of Proposition 2, the lemma holds. □

C Computing Successors in Weighted Pushdown Systems

In Section 3, we considered the following problem: given a *target* configuration c_f, compute (if possible) the meet-over-all-paths from c to c_f, for any configuration c. In other words, we considered the *predecessors* of c_f.

Alternatively, one could consider the *successors* of some *source* configuration $c_s := p_s X_s$ and attempt to compute the meet over all paths from c_s to c. It is possible to adapt the methods from Section 3 to this problem (and in fact, this adaptation is used by our applications).

It is well-known that most results about backward pushdown reachability carry over to forward pushdown reachability, and vice versa. The easiest explanation for this is that given a WPDS $\mathcal{W}$, one can construct another WPDS $\mathcal{W}'$ which makes the movements of $\mathcal{W}$ 'in reverse'. More precisely, if $\mathcal{W}$ has control states P, stack alphabet Γ, and rules Δ, then $\mathcal{W}'$ has control states $P' := P \cup \{\, (q, Y) \mid \exists (pX \stackrel{d}{\hookrightarrow} qYZ) \in \Delta \,\}$, stack alphabet $\Gamma \cup \{\#\}$, and the following rules:

- if $pX \stackrel{d}{\hookrightarrow} qY \in \Delta$, then $qY \stackrel{d}{\hookrightarrow} pX \in \Delta'$;
- if $pX \stackrel{d}{\hookrightarrow} q\epsilon \in \Delta$, then $qY \stackrel{d}{\hookrightarrow} pXY \in \Delta'$ for every $Y \in \Gamma \cup \{\#\}$;
- if $pX \stackrel{d}{\hookrightarrow} qYZ \in \Delta$, then $qY \stackrel{\overline{1}}{\hookrightarrow} (q, Y)\epsilon$ and $(q, Y)Z \stackrel{d}{\hookrightarrow} pX$ in Δ'.

It is easy to see that whenever $p\alpha \stackrel{\sigma}{\Rightarrow} q\beta$ holds in $\mathcal{W}$, then $q\beta\# \stackrel{\tau}{\Rightarrow} p\alpha\#$ holds for some rule sequence τ in $\mathcal{W}'$ such that, if $\sigma = r_1 \ldots r_n$ and $\tau = s_1 \ldots s_m$, then $d_{r_1} \otimes \cdots \otimes d_{r_n} = d_{s_m} \otimes \cdots \otimes d_{s_1}$. Thus, it is possible to reduce forward reachability problems to backward reachability problems, and the reduction is polynomial.

It is also possible to tackle the forward reachability problem directly, in which case slightly better complexity bounds can be achieved, see, for instance [3, 19]. Following the ideas from [3, 19], we will present a finite equation system that serves as the 'forward analogy' of (2), without proof. Our system has the following sets of variables:

- $[pX\bullet]$, for $p \in P$ and $X \in \Gamma$, representing the weights of the paths from $p_s X_s$ to pX;

– $[pX(rZ)]$, for $p \in P$, $X \in \Gamma$, and $(r, Z) \in P'$, representing the weights of the paths from rZ to pX;
– $[p\epsilon\bullet]$, for $p \in P$, representing the weights of the paths from $p_s X_s$ to $p\epsilon$;
– $[p\epsilon(rZ)]$, for $p \in P$ and $(r, Z) \in P'$, representing the weights of the paths from rZ to $p\epsilon$;
– $[(pX)Y\bullet]$, for $(p, X) \in P'$ and $Y \in \Gamma$, representing the weights of the paths from $p_s X_s$ to pXY, ending with a 'push' operation;
– $[(pX)Y(rZ)]$, for $(p, X), (r, Z) \in P'$ and $Y \in \Gamma$, representing the weights of the paths from rZ to pXY, ending with a 'push' operation.

Moreover, we define $I(pX) = \overline{1}$ iff $pX = p_s X_s$ and $\overline{0}$ otherwise, and $E(pX, rZ) = \overline{1}$ iff $pX = rZ$ and $\overline{0}$ otherwise, for $(p, X) \in P'$. The equation system is as follows:

$$[pX\bullet] = I(pX) \oplus \bigoplus_{qY \overset{d}{\hookrightarrow} pX} \big([qY\bullet] \otimes d\big) \oplus \bigoplus_{(q,Y)\in P'} \big([(qY)X\bullet] \otimes [p\epsilon(qY)]\big)$$

$$[pX(rZ)] = E(pX, rZ) \oplus \bigoplus_{qY \overset{d}{\hookrightarrow} pX} \big([qY(rZ)] \otimes d\big) \oplus \bigoplus_{(q,Y)\in P'} \big([(qY)X(rZ)] \otimes [p\epsilon(qY)]\big)$$

$$[p\epsilon\bullet] = \bigoplus_{qY \overset{d}{\hookrightarrow} p\epsilon} \big([qY\bullet] \otimes d\big)$$

$$[p\epsilon(rZ)] = \bigoplus_{qY \overset{d}{\hookrightarrow} p\epsilon} \big([qY(rZ)] \otimes d\big)$$

$$[(pX)Y\bullet] = \bigoplus_{qU \overset{d}{\hookrightarrow} pXY} \big([qU\bullet] \otimes d\big)$$

$$[(pX)Y(rZ)] = \bigoplus_{qU \overset{d}{\hookrightarrow} pXY} \big([qU(rZ)] \otimes d\big)$$

Intuitively, the right-hand sides of the equations list the possible ways in which the paths corresponding to the left-hand-side variables can be generated.

In analogy with Section 3.1, any solution of $\boldsymbol{h}$ can be converted into a $post^*(\{p_s X_s\})$ $\mathcal{W}$-automaton. Our automaton $\mathcal{A}$ has ϵ-edges, and its states are P' extended with a final state $\bullet$. Every variable $[sXs']$, where $s, s' \in P' \cup \{\bullet\}$ and $X \in \Gamma \cup \{\epsilon\}$, and its value in the solution then correspond to a transition of $\mathcal{A}$. The meet-over-all-paths for every configuration c can be obtained by identifying the paths on which c is accepted by $\mathcal{A}$ and computing $v_{\mathcal{A}}(c)$.

Remark 4. According to [20, 19], the size of the equation system and the number of variables is $\mathcal{O}(|P| \cdot |\Delta|^2)$, therefore the time for Algorithm 1 is $\mathcal{O}(|P|^2 |\Delta|^4)$. The resulting automaton has got $\mathcal{O}(|P| + |\Delta|)$ states.