Program Analysis Using Weighted Pushdown Systems*

Thomas Reps, Akash Lal, and Nick Kidd

Comp. Sci. Dept., University of Wisconsin; {reps, akash, kidd}@cs.wisc.edu

Abstract. *Pushdown systems* (PDSs) are an automata-theoretic formalism for specifying a class of infinite-state transition systems. Infiniteness comes from the fact that each configuration $\langle p, S \rangle$ in the state space consists of a (formal) "control location" p coupled with a stack S of unbounded size. PDSs can model program paths that have matching calls and returns, and automaton-based representations allow analysis algorithms to account for the infinite control state space of recursive programs.

Weighted pushdown systems (WPDSs) are a generalization of PDSs that add a general "black-box" abstraction for program data (through *weights*). WPDSs also generalize other frameworks for interprocedural analysis, such as the Sharir-Pnueli functional approach.

This paper surveys recent work in this area, and establishes a few new connections with existing work.

1 Introduction

Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Static-analysis techniques explore the program's behavior for *all* possible inputs and account for *all* possible states that the program can reach. In this sense, static analysis is more comprehensive than traditional testing, which tests the program's behavior for a fixed (possibly randomly generated) finite set of runs of the program. For any non-trivial program, it is impossible to test explicitly all the possible behaviors within a reasonable amount of time; in contrast, static-analysis techniques use *approximations* to account for all of the actions that the program could perform [13]. To make this feasible, two techniques are used:

- The program is *run in the aggregate*. Rather than executing the program on ordinary states, the program is executed on finite-sized descriptors that represent collections of states.
- The program is *run in a non-standard fashion*. Rather than executing the program in a linear sequence, various fragments are executed (in the aggregate) so that, when stitched together, the results are guaranteed to cover all possible execution paths.

* Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

Analysis algorithms typically use the program's interprocedural control-flow graph (also known as its *ICFG*). An ICFG consists of a collection of control-flow graphs (CFGs)—one for each procedure—one of which represents the program's main procedure. The CFG for a procedure p has a unique *enter* node and a unique *exit* node. The other nodes represent the program's statements and conditions (or, alternatively, its basic blocks), except that each procedure call in the program is represented in the ICFG by two nodes, a *call* node and a *return-site* node. *Call-edges* connect call nodes to enter nodes; *return-edges* connect exit nodes to return-site nodes. A typical analysis goal is to compute, for each ICFG node n, an overapproximation (i.e., superset) of the set of states that can hold when n is reached.

The choice of which family of data descriptors that an algorithm uses impacts which behavioral properties of the program can be observed. This, in turn, affects (i) what sets of states can be represented, and (ii) which program fragments need to be explored. For example, one might use descriptors that represent only the sign of a variable's value: `neg`, `zero`, `pos`, and `unknown`. In a context in which it is known that both `a` and `b` are positive (i.e., when the memory descriptor is $\langle \texttt{a} \mapsto \texttt{pos}, \texttt{b} \mapsto \texttt{pos} \rangle$), a multiplication expression such "`a*b`" would be performed as "`pos*pos`".

Such memory descriptors generally represent a superset of the actual set of memory states that are reachable, because a descriptor such as $\langle \texttt{a} \mapsto \texttt{pos}, \texttt{b} \mapsto \texttt{pos} \rangle$ represents all states in which `a` and `b` hold positive integers (whereas, for example, only combinations with odd positive `a`'s and even positive `b`'s might be reachable). At a branch-point in the program, the analyzer needs to observe the possible outcomes of the branch-point's condition—as best it can, given the memory descriptors in use. This is used to determine an overapproximation of the paths along which control might flow. Thus, a more refined class of data descriptors can sometimes allow certain paths to be excluded from consideration.

On the other hand, certain paths can be excluded merely from consideration of the control-flow properties of the programming language. An important class of paths that can be excluded are those that violate the language's call/return protocol; in particular, an analysis should only consider paths in which the return from a called procedure is matched with the most recent call. Fig. 1 shows a fragment of an ICFG, and an example of a path fragment that should be excluded from consideration.

Dataflow-analysis algorithms that exclude such paths have a long history [14, 47, 26]. A natural class of dataflow-analysis problems in which this issue is reduced to a pure graph-reachability problem is also known [40]. The algorithms developed for that class of problems are useful for analyzing a family of program abstractions called Boolean programs (§2.3). (Boolean programs have become well-known due to their use in SLAM [4, 5] to represent program abstractions obtained via predicate abstraction [20].)

More recently, analysis techniques based on pushdown systems (PDSs) [6, 18, 44] have been developed. PDSs are an automata-theoretic formalism for speci-

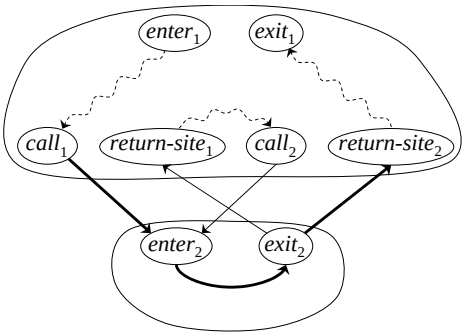


Fig. 1. An invalid-path fragment: in the path $[call_1, enter_2, exit_2, return\text{-}site_2]$, the return-edge $exit_2 \rightarrow return\text{-}site_2$ does not match with call-edge $call_1 \rightarrow enter_2$.

fying a class of infinite-state transition systems. Infiniteness comes from the fact that each configuration $\langle p, S \rangle$ in the state space consists of a (formal) "control location" p coupled with a stack S of unbounded size. Boolean programs have natural encodings as PDSs (see §2.3). Moreover, techniques developed for answering reachability queries on PDSs allow dataflow queries to be posed with respect to a *regular language of configurations*, which allows one to recover dataflow information for specific *calling contexts* (and for regular languages of calling contexts).

Subsequently, these techniques were generalized to *Weighted Pushdown Systems* (WPDSs) [7, 46, 41, 42]. WPDSs extend PDSs by adding a general "black-box" abstraction for expressing transformations of a program's data state (through *weights*). By extending methods from PDSs that answer questions about only certain sets of paths (namely, ones that end in a specified regular language of configurations), WPDSs generalize other frameworks for interprocedural analysis, such as the Sharir-Pnueli functional approach [47], as well as the Knoop-Steffen [26] and Sagiv-Reps-Horwitz summary-based approaches [43]. In particular, conventional dataflow-analysis algorithms merge together the values for all states associated with the same program point, regardless of the states' calling context.

Because WPDSs permit dataflow queries to be posed with respect to a regular language of stack configurations,[1] one obtains several benefits from recasting an existing dataflow-analysis algorithm into the WPDS framework. First, one immediately obtains algorithms to find dataflow information for specific calling contexts and families of calling contexts, which provides information that was not previously obtainable. For instance, §3.2 and §4 discuss, respectively, how to recast Müller-Olm and Seidl's work on affine-relation analysis [34, 35] and Landi and Ryder's work on may-aliasing for single-level pointer programs [32] in the WPDS framework, which makes it possible to pose stack-qualified queries about

[1] 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 prior approaches.

affine relations and may-alias relations. Second, the algorithms for solving path problems in WPDSs can provide a witness set of paths [42], which is useful for providing an explanation of why the answer to a dataflow query has the value reported.

Two implementations of WPDSs are publicly available [45, 24], and both provide a convenient base for implementing different analyses. As a programming abstraction, these systems offer several benefits:

- An analyzer is created by means of a declarative specification: one specifies a weight domain, along with an encoding of the program's ICFG and a mapping of each ICFG edge to a weight.
- It permits the creation of libraries of reusable weight domains, which can also be used to create new weight domains by means of weight-domain-construction operations (pairing, reduced product [15], tensor product [37], etc.)
- Advances in solver technology apply to all instantiations of the framework; for instance, Lal and Reps achieved substantial speedups over previous algorithms by using more sophisticated algorithms in the WPDS solver engine [29].

WPDS++ [24] has been used to implement several of the analyses in CodeSurfer/x86 [3, 30, 1], a system for analyzing Intel x86 executables. It has also been used as a core analysis component in a system for analyzing concurrent programs [12].

Compared with other tools that support the creation of program analyzers from high-level specifications, (i) the WPDS implementations allow more sophisticated abstract domains to be used (such as the Müller-Olm/Seidl domains for affine-relation analysis [34, 35]), and also permit a broader range of dataflow-analysis queries to be posed than is possible with Banshee [27] and BDDBDDB [48]; (ii) the WPDS implementations support a broader range of dataflow-analysis queries than PAG [33].

Organization of the Paper. This paper surveys our recent work on WPDSs, and establishes a few new connections with other work. The remainder of the paper is organized into four sections: §2 provides background material on interprocedural dataflow analysis, PDSs, and Boolean programs. §3 introduces WPDSs. §4 describes how the work of Landi and Ryder [32] on single-level pointer analysis can be expressed in the WPDS framework. §5 summarizes recent work both on improving and on applying WPDS technology.

2 Background

2.1 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 CFG of the program.
– A meet semilattice $(V, \sqcap)$ with greatest element $\top$:
  - An element of V represents a set 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 M of dataflow transfer functions (of type $V \to V$) to the edges of the CFG: $M(e) \in V \to V$.

A dataflow-analysis problem can be formulated as a *path-function problem*.

Definition 1. *A* **path** *of length j from node m to node n is a (possibly empty) sequence of j edges, 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}.*

The path function pf_q for path $q = [e_1, e_2, \ldots, e_j]$ is the composition, in order, of q's transfer functions: $\mathrm{pf}_q = M(e_j) \circ \ldots \circ M(e_2) \circ M(e_1)$. 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 $\mathrm{Paths}(\mathrm{enter}, n)$ denotes the set of paths in the CFG from the enter node to n [25]. 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.

The soundness of the MOP_n solution with respect to the programming language's concrete semantics is established by the methodology of *abstract interpretation* [13]:

– A Galois connection (or Galois insertion) is established to define the relationship between sets of concrete states and elements of V.
– Each dataflow transfer function $M(e)$ is shown to overapproximate the transfer function for the concrete semantics of e.

In this paper, we assume that such correctness requirements have already been taken care of; the paper concentrates on algorithms for determining dataflow values once an instance of a dataflow-analysis problem has been given.

An example ICFG is shown in Fig. 2. Let *Var* be the set of all variables in a program, and let $(\mathbb{Z}_\perp, \sqsubseteq, \sqcap)$, where $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$, be the standard constant-propagation semilattice: for all $c \in \mathbb{Z}$, $\perp \sqsubset c$; for all $c_1, c_2 \in \mathbb{Z}_\perp$ such that $c_1 \neq c_2$, c_1 and c_2 are incomparable; and $\sqcap$ is the greatest-lower-bound operation in this partial order. $\perp$ stands for "not-a-constant". Let $D = (\mathit{Env} \to \mathit{Env})$ be the set of all environment transformers where an environment is a mapping for all variables: $\mathit{Env} = (\mathit{Var} \to \mathbb{Z}_\perp) \cup \{\top\}$. We use $\top$ to denote an infeasible environment. Furthermore, we restrict the set D to contain only $\top$-strict transformers, i.e.,

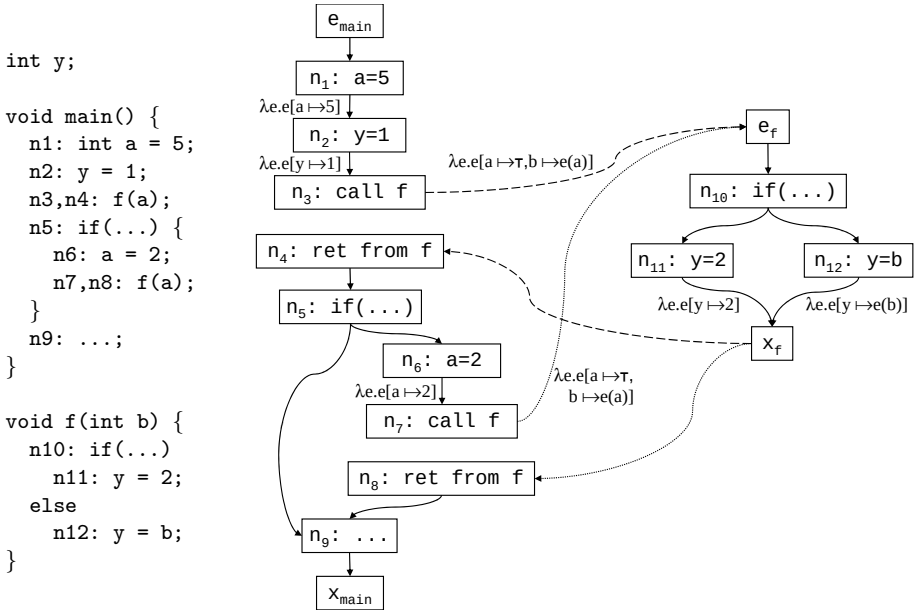


Fig. 2. A program fragment and its ICFG. For all unlabeled edges, the environment transformer is $\lambda e.e$.

for all $d \in D$, $d(\top) = \top$. We can extend the meet operation to environments by taking meet componentwise.

$$env_1 \sqcap env_2 = \begin{cases} env_1 & \text{if } env_2 = \top \\ env_2 & \text{if } env_1 = \top \\ \lambda v.(env_1(v) \sqcap env_2(v)) & \text{otherwise} \end{cases}$$

The dataflow transformers are shown as edge labels in Fig. 2. A transformer of the form $\lambda e.e[a \mapsto 5]$ returns an environment that agrees with the argument, except that a is bound to 5. The environment $\top$ cannot be updated, and thus $(\lambda e.e[a \mapsto 5])\top$ equals $\top$.

The notion of an *(interprocedurally) valid path* captures the idea that not all paths in an ICFG 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. Let each call node in the ICFG be given a unique index from 1 to *CallSites*, where *CallSites* is the total number of call sites in the program. For each call site c_i, label the call-to-enter edge and the exit-to-return-site edge with the symbols "$(_i$" and "$)_i$", respectively. Label all other edges of the ICFG with the symbol e. Each path in the ICFG defines a word, obtained by concatenating—in order—the labels of the edges on the path. A path is a *valid path* iff the path's word is in the language $L(valid)$ generated by the context-free grammar shown below on the left; a path is a *matched path* iff the path's word is in the language $L(matched)$ of balanced-parenthesis strings (interspersed with strings of zero or

more e's) generated by the context-free grammar shown below on the right. (In both grammars, i ranges from 1 to *CallSites*.)

$$\begin{array}{lll} \mathit{valid} \rightarrow \mathit{matched}\ \ \mathit{valid} & \qquad & \mathit{matched} \rightarrow \mathit{matched}\ \ \mathit{matched} \\ \quad \mid\ (_i\ \ \mathit{valid} & & \quad \mid\ (_i\ \ \mathit{matched}\ \ )_i \\ \quad \mid\ \epsilon & & \quad \mid\ e \\ & & \quad \mid\ \epsilon \end{array}$$

The language $L(\mathit{valid})$ is a language of *partially* balanced parentheses: every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse need not hold.

Example 1. In the ICFG shown in Fig. 2, the path $[e_{main}, n_1, n_2, n_3, e_f, n_{10}, n_{11}, x_f, n_4, n_5]$ is a matched path, and hence a valid path; the path $[e_{main}, n_1, n_2, n_3, e_f, n_{10}]$ is a valid path, but not a matched path, because the call-to-enter edge $n_3 \rightarrow e_f$ has no matching exit-to-return-site edge; the path $[e_{main}, n_1, n_2, n_3, e_f, n_{10}, n_{11}, x_f, n_8]$ is neither a matched path nor a valid path because the exit-to-return-site edge $x_f \rightarrow n_8$ does not correspond to the preceding call-to-enter edge $n_3 \rightarrow e_f$.

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 only on the memory configurations that arise at the call site that corresponds 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 ICFG [47, 26, 43]:

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

where $\mathrm{VPaths}(\mathrm{e_{main}}, n)$ 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 $\mathrm{e_{main}}$, 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.

2.2 Pushdown Systems

In this section, we define pushdown systems and show how they can be used to encode ICFGs.

Definition 2. *A* ***pushdown system*** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of* states *(also known as "control locations"), Γ is a finite set of stack*

Rule	Control flow modeled
$\langle p, u\rangle \hookrightarrow \langle p, v\rangle$	Intraprocedural edge $u \to v$
$\langle p, c\rangle \hookrightarrow \langle p, e_f\ r\rangle$	Call to f from c that returns to r
$\langle p, x_f\rangle \hookrightarrow \langle p, \varepsilon\rangle$	Return from f at exit node x_f

Fig. 3. The encoding of an ICFG's edges as PDS rules.

*symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A **configuration** of $\mathcal{P}$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u'\rangle$, then $\langle p, \gamma u\rangle \Rightarrow \langle p', u'u\rangle$ for all $u \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under the transition relation $\Rightarrow$.*

Without loss of generality, we restrict the pushdown rules to have at most two stack symbols on the right-hand side [44]. A rule $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, $u \in \Gamma^*$, is called a *pop* rule if $|u| = 0$, and a *push* rule if $|u| = 2$.

The PDS configurations model (node, stack) pairs of the program's state. Given a program P, we can use a PDS to model a limited portion of a P's behavior in the following sense: the configurations of the PDS represent a superset of P's (node, stack) pairs.

The standard approach for modeling a program's control flow with a pushdown system is as follows: P contains a single state p, Γ corresponds to the nodes of the program's ICFG, and Δ corresponds to edges of the program's ICFG (see Fig. 3). For instance, the rules that encode the ICFG shown in Fig. 2 are

$$\begin{array}{lll}
\langle p, e_{main}\rangle \hookrightarrow \langle p, n_1\rangle & \langle p, n_5\rangle \hookrightarrow \langle p, n_9\rangle & \langle p, e_f\rangle \hookrightarrow \langle p, n_{10}\rangle \\
\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle & \langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle & \langle p, n_{10}\rangle \hookrightarrow \langle p, n_{11}\rangle \\
\langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle & \langle p, n_7\rangle \hookrightarrow \langle p, e_f\ n_8\rangle & \langle p, n_{11}\rangle \hookrightarrow \langle p, x_f\rangle \\
\langle p, n_3\rangle \hookrightarrow \langle p, e_f\ n_4\rangle & \langle p, n_8\rangle \hookrightarrow \langle p, n_9\rangle & \langle p, n_{10}\rangle \hookrightarrow \langle p, n_{12}\rangle \\
\langle p, n_4\rangle \hookrightarrow \langle p, n_5\rangle & \langle p, n_9\rangle \hookrightarrow \langle p, x_{main}\rangle & \langle p, n_{12}\rangle \hookrightarrow \langle p, x_f\rangle \\
\langle p, n_5\rangle \hookrightarrow \langle p, n_6\rangle & \langle p, x_{main}\rangle \hookrightarrow \langle p, \varepsilon\rangle & \langle p, x_f\rangle \hookrightarrow \langle p, \varepsilon\rangle
\end{array}$$

PDSs that have only a single control location, as discussed above, are also called "context-free processes" [10]. In §2.3, we will discuss how, in addition to control flow, PDSs can also be used to encode program models that involve finite abstractions of the program's data. PDSs that have multiple control locations are used in such encodings.

The problem of interest is to find the set of all reachable configurations, starting from a given set of configurations. This can then be used, for example, for assertion checking (i.e., determining if a given assertion can ever fail) or to find the set of all data values that may arise at a program point (for dataflow analysis).

Because the number of configurations of a pushdown system is unbounded, it is useful to use finite automata to describe regular sets of configurations.

Definition 3. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS then a* **$\mathcal{P}$-automaton** *is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states. We say that a configuration $\langle p, u \rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in the state p (written as $p \xrightarrow{u}{}^{*} q$, where $q \in F$). A set of configurations is called* ***regular*** *if some $\mathcal{P}$-automaton accepts it. Without loss of generality, $\mathcal{P}$-automata are restricted to not have any transitions leading to an initial state.*

An important result is that for a regular set of configurations C, both $post^*(C)$ and $pre^*(C)$ (the forward and the backward reachable sets of configurations, respectively) are also regular sets of configurations [6, 9]. The algorithms for computing $post^*$ and pre^*, called *poststar* and *prestar*, respectively, take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if C is the set of configurations accepted by $\mathcal{A}$, they produce $\mathcal{P}$-automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ that accept the sets of configurations $post^*(C)$ and $pre^*(C)$, respectively [6, 17, 18]. Both *poststar* and *prestar* can be implemented as *saturation procedures*; i.e., transitions are added to $\mathcal{A}$ according to some saturation rule until no more can be added.

Algorithm *prestar***:** $\mathcal{A}_{pre^*}$ can be constructed from $\mathcal{A}$ using the following saturation rule: *If $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and $p' \xrightarrow{w} q$ in the current automaton, add a transition (p, γ, q).*

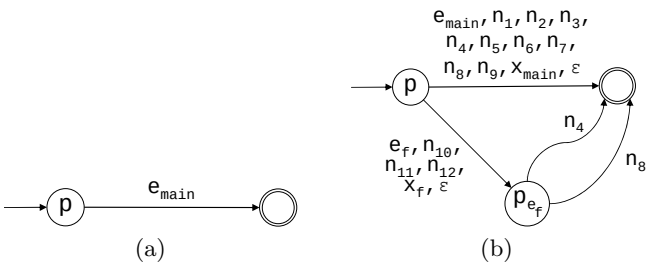
Algorithm *poststar***:** $\mathcal{A}_{post^*}$ can be constructed from $\mathcal{A}$ by performing Phase I and then saturating via the rules given in Phase II:

- *Phase I.* For each pair (p', γ') 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 $p'_{\gamma'}$.
- *Phase II (saturation phase).* (The symbol $\overset{\gamma}{\rightsquigarrow}$ denotes the relation $(\xrightarrow{\epsilon})^\star \xrightarrow{\gamma} (\xrightarrow{\epsilon})^\star$.)
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$ 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, add the transitions $(p', \gamma', p'_{\gamma'})$ and $(p'_{\gamma'}, \gamma'', q)$.

Example 2. Given the PDS that encodes the ICFG from Fig. 2 and the query automaton $\mathcal{A}$ shown in Fig. 4(a), which accepts the language $\{\langle p, e_{main} \rangle\}$, *poststar* produces the automaton $\mathcal{A}_{post^*}$ shown in Fig. 4(b),

2.3 Boolean Programs

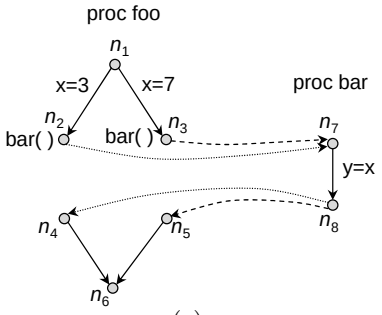
A Boolean program can be thought of as a C program with only the Boolean datatype. It does not have any pointers or heap-allocated storage. A Boolean



(a) (b)

Fig. 4. (a) Automaton for the input language of configurations $\{\langle p, e_{main}\rangle\}$; (b) automaton for $post^*(\{\langle p, e_{main}\rangle\})$ (computed for the PDS that encodes the ICFG from Fig. 2).

program consists of a finite set of procedures. It has a finite set of global variables, and a finite set of local variables for each procedure. Each variable can only hold a value from a finite domain.[2] To simplify the discussion, we assume that procedures do not have parameters (they can be passed through global variables). The variables in scope inside a procedure are the global variables and its set of local variables. Fig. 5(a) shows a Boolean program with two procedures and two global variables `x` and `y` over a finite domain $V = \{0, 1, \ldots, 7\}$.



(a)

$$[\![x = 3]\!] = \{((v_1, v_2), (3, v_2)) \mid v_1, v_2 \in V\}$$
$$[\![x = 7]\!] = \{((v_1, v_2), (7, v_2)) \mid v_1, v_2 \in V\}$$
$$[\![y = x]\!] = \{((v_1, v_2), (v_1, v_1)) \mid v_1, v_2 \in V\}$$

(b)

Fig. 5. (a) A Boolean program with two procedures and two global variables `x` and `y` over a finite domain $V = \{0, 1, \ldots, 7\}$. (b) The (non-identity) transformers used in the Boolean program.

Notation. A binary relation on a set S is a subset of $S \times S$. If R_1 and R_2 are binary relations on S, then their relational composition, denoted by "$R_1; R_2$", is defined by $\{(s_1, s_3) \mid \exists s_2 \in S, (s_1, s_2) \in R_1, (s_2, s_3) \in R_2\}$. If R is a binary relation, R^i is the relational composition of R with itself i times, and R^0 is the identity relation on S. $R^* = \cup_{i=0}^{\infty} R^i$ is the reflexive-transitive closure of R.

[2] An assignment to a variable v that holds a value from a finite domain can be thought of a collection of assignments to a *vector* of Boolean-valued variables, namely, the collection of Boolean-valued variables that holds the encoding of v's value.

Let G be the set of valuations of the global variables, and let Val_i be the set of valuations of the local variables of procedure i. Let L be the set of local states of the program; each local state consists of the value of the program counter, a valuation of local variables from some Val_i, and the program stack (which, for each unfinished call to a procedure P, contains a return address and a valuation of the local variables of P).

The effect of executing an assignment or assume statement `st`, denoted by $[\![\texttt{st}]\!]$, is a binary relation on $G \times \mathrm{Val}_i$ that describes how values of variables in scope can change. Fig. 5(b) shows the (non-identity) transformers used in Fig. 5(a).

To encode a Boolean program using a PDS, the state alphabet P is expanded to encode the values of global variables, and the stack alphabet is expanded to encode the values of local variables [44].

Let N_i be the set of control locations of the i^{th} procedure. We set P to be G, and Γ to be the union of $N_i \times \mathrm{Val}_i$ over all procedures. (Note that the set of local states L equals Γ^*.) The PDS rules for the i^{th} procedure are constructed as follows: (i) an intraprocedural ICFG edge $u \to v$ with action `st` is encoded via a set of rules $\langle g, (u, l)\rangle \hookrightarrow \langle g', (v, l')\rangle$, for each $((g, l), (g', l')) \in [\![\texttt{st}]\!]$; (ii) a call edge $c \to r$ that calls procedure f, with enter node e_f, is encoded via a set of rules $\langle g, (c, l)\rangle \hookrightarrow \langle g, (e_f, l_0)\ (r, l)\rangle$, for each $(g, l) \in G \times \mathrm{Val}_i$ and $l_0 \in \mathrm{Val}_f$; (iii) a procedure return at node u is encoded via a set of rules $\langle g, (u, l)\rangle \hookrightarrow \langle g, \varepsilon\rangle$, for each $(g, l) \in G \times \mathrm{Val}_i$;

Under such an encoding of a Boolean program as a PDS, a configuration $\langle p, \gamma_1\gamma_2 \cdots \gamma_n\rangle$ is an element of $G \times L$ that describes the instantaneous state of a program. The state p encodes the values of global variables; γ_1 encodes the current program location and the values of local variables in scope; and the rest of the stack encodes the list of unfinished calls with the values of local variables at the time the call was made. The PDS transition relation ($\Rightarrow$), which is essentially a transition relation on $G \times L$, represents the semantics of the Boolean program.

3 Weighted Pushdown Systems

A weighted pushdown system is obtained by augmenting a PDS with a weight domain that is a *bounded idempotent semiring* [42, 7]. Such semirings are powerful enough to encode finite-state data abstractions, such as the ones required for bitvector dataflow analysis, Boolean programs, and the IFDS framework of Reps et al. [40], as well as infinite-state data abstractions, such as linear-constant propagation [43] and affine-relation analysis [34, 35]. We present some of this here; additional material about using WPDSs for interprocedural analysis can be found in [42].

Weights encode the effect that each statement (or PDS rule) has on the data state of the program. They can be thought of as abstract transformers that specify how the abstract state changes when a statement is executed.

Definition 4. *A* **bounded idempotent semiring** *(or* **weight domain***) is a tuple* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, where* D *is a set whose elements are called* **weights***,*

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

Definition 5. *A* **weighted pushdown system** *is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$*, where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a PDS,* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *is a bounded idempotent semiring, and* $f : \Delta \to D$ *is a map that assigns a weight to each rule of* $\mathcal{P}$.

WPDSs compute over the weights via the extend operation ($\otimes$). Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ; i.e., if $\sigma = [r_1, \ldots, r_k]$, we define $v(\sigma) \stackrel{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. In program-analysis problems, weights typically represent abstract transformers that specify how the abstract state changes when a statement is executed. Thus, the extend operation is typically the reversal of function composition: $w_1 \otimes w_2 = w_2 \circ w_1$. (Computing over transformers by composing them—instead of computing on the underlying abstract states by applying transformers to abstract states—is customary in interprocedural analysis, where procedure summaries need to be calculated as compositions of abstract-state transformers [14, 26, 40].)

Reachability problems on PDSs are generalized to WPDSs as follows:

Definition 6. *Let* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *be a weighted pushdown system, where* $\mathcal{P} = (P, \Gamma, \Delta)$*. For any two configurations* c *and* c' *of* $\mathcal{P}$*, let* $path(c, c')$ *denote the set of all rule sequences that transform* c *into* c'*. Let* $S, T \subseteq P \times \Gamma^*$ *be regular sets of configurations. If* $\sigma \in path(c, c')$*, then we say* $c \Rightarrow^\sigma c'$*. The* ***meet-over-all-valid-paths*** *value* $\text{MOVP}(S, T)$ *is defined as* $\bigoplus\{v(\sigma) \mid s \Rightarrow^\sigma t, s \in S, t \in T\}$.

A PDS, as defined in §2.2, is simply a WPDS with the *Boolean weight domain* $(\{F, T\}, \vee, \wedge, F, T)$ and weight assignment $f(r) = T$ for all rules $r \in \Delta$. In this case, $\text{MOVP}(S, U) = T$ iff there exists a path from a configuration in S to a configuration in U, i.e., $post^*(S) \cap U$ and $S \cap pre^*(U)$ are non-empty sets.

One way of modeling a program as a WPDS is as follows: the PDS models the control flow of the program, as in Fig. 3. The weight domain models abstract transformers for an abstraction of the program's data. §3.1 and §3.2 describe several data abstractions that can be encoded using weight domains. To simplify the presentation, we only show the treatment for global variables, and do not consider local variables. Finite-state abstractions of local variables can always be encoded in the stack alphabet, as for PDSs [30, 44]. For infinite-state abstractions, local variables pose an extra complication for WPDSs [30]; their treatment is discussed in §3.4.

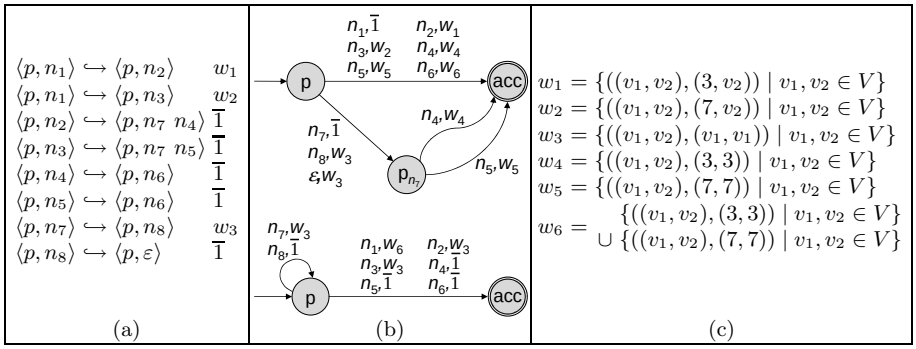


Fig. 6. (a) A WPDS that encodes the Boolean program from Fig. 5(a). (b) The result of *poststar*($\langle p, n_1 \rangle$) and *prestar*($\langle p, n_6 \rangle$). The final state in each of the automata is *acc*. (c) Definitions of the weights used in the figure.

3.1 Finite-State Data Abstractions

An important weight domain for WPDSs is the set of all binary relations on a finite set.

Definition 7. *If G is a finite set, then the* ***relational weight domain*** *on G is defined as $(2^{G \times G}, \cup, ;, \emptyset, id)$: weights are binary relations on G, combine is union, extend is relational composition (";"), $\overline{0}$ is the empty relation, and $\overline{1}$ is the identity relation on G.*

By instantiating G to be the set of global states of a Boolean program P, we obtain a weight domain for encoding P. This approach yields a more straightforward encoding of P: the weight associated with the rule that encodes an assignment or assume statement `st` of P is exactly $[\![\texttt{st}]\!]$—i.e., its effect on the global state of P—which, as described in §2.3, is a binary relation on G. For example, the WPDS shown in Fig. 6 encodes the Boolean program from Fig. 5(a). The Boolean program has two variables that range over the set $V = \{0, 1, \ldots, 7\}$, so $G = V \times V$, where the two components represent the values of `x` and `y`, respectively.

The set of all data values that reach a node n can be calculated as follows: let S be the singleton configuration consisting of the program's enter node, and let T be the set $\{\langle p, n\ u \rangle \mid u \in \Gamma^*\}$. Let $w = \text{MOVP}(S, T)$. If $w = \overline{0}$, then the node cannot be reached. Otherwise, w captures the net transformation on the global state from when the program started. The range of w, i.e., the set $\{g \in G \mid \exists g' \in G : (g', g) \in w\}$, is the set of valuations that reach node n. For example, in Fig. 6, the MOVP weight to node n_6 is the weight w_6 shown in Fig. 6(c). Its range shows that either $\texttt{x} = 3$ and $\texttt{y} = 3$, or $\texttt{x} = 7$ and $\texttt{y} = 7$.

Because T can be any regular set, one can also answer stack-qualified queries [42]. For example, the set of values that arise at node n when its procedure is called from call site m can be found by setting $T = \{\langle p, n\ m_r\ u \rangle \mid u \in \Gamma^*\}$, where m_r is the return site for call site m.

A WPDS with a weight domain that has a finite set of weights, such as the one described above, can be encoded as a PDS. However, it is often useful to use weights because they can be symbolically encoded. Tools such as MOPED and SLAM use BDDs [8] to encode sets of data values, which allows them to scale to a large number of variables. (Using PDSs for Boolean program verification, without any symbolic encoding, is generally not a feasible approach.)

3.2 Infinite-State Data Abstractions

An infinite-state data abstraction is one in which the number of abstract states (or weights) is infinite. We begin with two simple examples of infinite weight domains, and then discuss the weight domain used for affine-relation analysis.

Finding Shortest Valid Paths.

Definition 8. *The **minpath semiring** is the weight domain* $\mathcal{M} = (\mathbb{N} \cup \{\infty\}, min, +, \infty, 0)$*: weights are non-negative integers including "infinity", combine is minimum, and extend is addition.*

If all rules of a WPDS are given the weight 1 from this semiring (different from the semiring weight $\overline{1}$, which is the integer 0), then the MOVP weight between two configurations is the length of the shortest path (shortest rule sequence) between them.

Another infinite weight domain, which is based on the minpath semiring, is given in [28] and was shown to be useful for debugging programs.

Finding Shortest Traces. The minpath semiring can be combined with a relational weight domain, for example, to find the shortest (valid) path in a Boolean program (for finding the shortest trace that exhibits some property).

Definition 9. *A **weighted relation** on a set* S*, weighted with semiring* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, is a function from* $(S \times S)$ *to* D*. The composition of two weighted relations* R_1 *and* R_2 *is defined as* $(R_1; R_2)(s_1, s_3) = \oplus\{w_1 \otimes w_2 \mid \exists s_2 \in S : w_1 = R_1(s_1, s_2), w_2 = R_2(s_2, s_3)\}$*. The union of the two weighted relations is defined as* $(R_1 \cup R_2)(s_1, s_2) = R_1(s_1, s_2) \oplus R_2(s_1, s_2)$*. The identity relation is the function that maps each pair* (s, s) *to* $\overline{1}$ *and others to* $\overline{0}$*. The reflexive transitive closure is defined in terms of these operations, as before. If* $\rightarrow$ *is a weighted relation and* $(s_1, s_2, w) \in \rightarrow$*, then we write* $s_1 \xrightarrow{w} s_2$*.*

Definition 10. *If* $\mathcal{S}$ *is a weight domain with set of weights* D *and* G *is a finite set, then the relational weight domain on* $(G, \mathcal{S})$ *is defined as* $(2^{G \times G \rightarrow D}, \cup, ;, \emptyset, id)$*: weights are weighted relations on* G *and the operations are the corresponding ones for weighted relations.*

If G is the set of global states of a Boolean program, then the relational weight domain on $(G, \mathcal{M})$ can be used for finding the shortest trace: for each

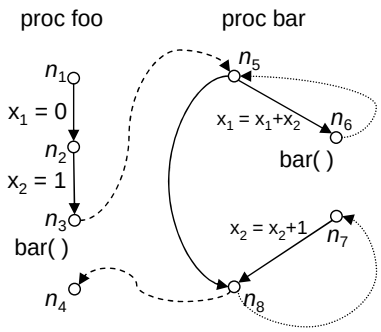


Fig. 7. An affine program that starts execution at node $\mathtt{n}_1$. There are two global variables $\mathtt{x}_1$ and $\mathtt{x}_2$.

rule, if $R \subseteq G \times G$ is the effect of executing the rule on the global state of the Boolean program, then associate the following weight with the rule:

$$\{g_1 \xrightarrow{1} g_2 \mid (g_1, g_2) \in R\} \cup \{g_1 \xrightarrow{\infty} g_2 \mid (g_1, g_2) \notin R\}.$$

Then, if $w = \mathrm{MOVP}(C_1, C_2)$, the length of the shortest path that starts with global state g from a configuration in C_1 and ends at global state g' in a configuration in C_2, is $w(g, g')$ (which would be ∞ if no path exists). (Moreover, if a finite-length path does exist, a witness trace [42] can be obtained to identify the elements of the path.)

Affine-Relation Analysis An affine relation is a linear-equality constraint between integer-valued variables. Affine-relation analysis (ARA) tries to find all affine relationships that hold in the program. An example is shown in Fig. 7. For this program, ARA would, for example, infer that $\mathtt{x}_2 = \mathtt{x}_1 + 1$ at program node $\mathtt{n}_4$.

ARA for single-procedure programs was first given by Karr [23]. ARA generalizes other analyses, including copy-constant propagation, linear-constant propagation [43], and induction-variable analysis [23]. We have used ARA on machine code to find induction-variable relationships between machine registers [2]. These help in increasing the precision of an abstract-interpretation-based pointer analysis for machine code [1].

Affine Programs. Interprocedural ARA can be performed precisely on *affine programs*, and has been the focus of several papers [34, 35, 21]. Affine programs are similar to Boolean programs, but with integer-valued variables. Again, we restrict our attention to global variables, and defer treatment of local variables to §3.4. If $\{\mathtt{x}_1, \mathtt{x}_2, \cdots, \mathtt{x}_n\}$ is the set of global variables of the program, then all assignments have the form $\mathtt{x}_j := a_0 + \sum_{i=1}^{n} a_i \mathtt{x}_i$, where $a_0, \cdots, a_n$ are integer constants. An assignment can also be non-deterministic, denoted by $\mathtt{x}_j :=$?,

which may assign any integer to $\mathtt{x}_j$. (This is typically used for abstracting assignments that cannot be modeled as an affine transformation of the variables.) All branch conditions in affine programs are non-deterministic.

ARA Weight Domain. We briefly describe the weight domain based on the linear-algebra formulation of ARA from [34]. An affine relation $a_0 + \sum_{i=1}^{n} a_i \mathtt{x}_i = 0$ is represented using a column vector of size $n+1$: $\boldsymbol{a} = (a_0, a_1, \cdots, a_n)^t$. A valuation of program variables $\overline{\mathtt{x}}$ is a map from the set of global variables to the integers. The value of $\mathtt{x}_i$ under this valuation is written as $\overline{\mathtt{x}}(i)$.

A valuation $\overline{\mathtt{x}}$ satisfies an affine relation $\boldsymbol{a} = (a_0, a_1, \cdots, a_n)^t$ if $a_0 + \sum_{i=1}^{n} a_i \overline{\mathtt{x}}(i) = 0$. An affine relation $\boldsymbol{a}$ represents the set of all valuations that satisfy it, written as $\textsc{Pts}(\boldsymbol{a})$. An affine relation $\boldsymbol{a}$ holds at a program node if the set of valuations reaching that node (in the concrete collecting semantics) is a subset of $\textsc{Pts}(\boldsymbol{a})$.

An important observation about affine programs is that if affine relations $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ hold at a program node, then so does any linear combination of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$. For example, one can verify that $\textsc{Pts}(\boldsymbol{a}_1 + \boldsymbol{a}_2) \supseteq \textsc{Pts}(\boldsymbol{a}_1) \cap \textsc{Pts}(\boldsymbol{a}_2)$, i.e., the affine relation $\boldsymbol{a}_1 + \boldsymbol{a}_2$ (componentwise addition) holds at a program node if both $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ hold at that node. The set of affine relations that hold at a program node forms a (finite-dimensional) vector space [34]. This implies that a (possibly infinite) set of affine relations can be represented by any of its bases; each such basis is always a finite set.

For reasoning about affine programs, Müller-Olm and Seidl defined an abstraction that is able to find all affine relationships in an affine program: each statement is abstracted by a set of matrices of size $(n+1) \times (n+1)$. This set is the weakest-precondition transformer on affine relations for that statement: if a statement is abstracted as the set $\{m_1, m_2, \cdots, m_r\}$, then the affine relation $\boldsymbol{a}$ holds after the execution of the statement if and only if the affine relations $(m_1\boldsymbol{a}), (m_2\boldsymbol{a}), \cdots, (m_r\boldsymbol{a})$ held before the execution of the statement.

Under such an abstraction of program statements, one can define the extend operation, which is transformer composition, as elementwise matrix multiplication, and the combine operation as set union. This is correct semantically, but it does not give an effective algorithm because the matrix sets can grow unboundedly. However, the observation that affine relations form a vector space carries over to a set of matrices as well. One can show that the transformer $\{m_1, m_2, \cdots, m_r\}$ is semantically equivalent to the transformer $\{m_1, m_2, \cdots, m_r, m\}$, where m is any linear combination of the m_i matrices. Thus, a set of matrices can be abstracted as the (infinite) set of matrices spanned by them. Once we have a vector space, we can represent it using any of its bases to get a finite and bounded representation: a vector space over matrices of size $(n+1) \times (n+1)$ cannot have more that $(n+1)^2$ matrices in any basis.

If M is a set of matrices, let $\textsc{Span}(M)$ be the vector space spanned by them. Let β be the basis operation that takes a set of matrices and returns a basis of their span. We can now define the weight domain. A weight w is a vector space of matrices, which can be represented using its basis. Extend of vector spaces

w_1 and w_2 is the vector space $\{(m_1 m_2) \mid m_i \in w_i\}$. Combine of w_1 and w_2 is the vector space $\{(m_1 + m_2) \mid m_i \in w_i\}$, which is the smallest vector space containing both w_1 and w_2. $\overline{0}$ is the empty set, and $\overline{1}$ is the span of the singleton set consisting of the identity matrix. The extend and combine operations, as defined above, are operations on infinite sets. They can be implemented by the corresponding operations on any basis of the weights. The following properties show that it is semantically correct to operate on the elements in the basis instead of all the elements in the vector space spanned by them:

$$\beta(w_1 \oplus w_2) = \beta(\beta(w_1) \oplus \beta(w_2))$$
$$\beta(w_1 \otimes w_2) = \beta(\beta(w_1) \otimes \beta(w_2))$$

These properties are satisfied because of the linearity of extend (matrix multiplication distributes over addition) and combine operations.

Under such a weight domain, $\mathrm{MOVP}(S, T)$ is a weight that is the net weakest-precondition transformer between S and T. Suppose that this weight has the basis $\{m_1, \cdots, m_r\}$. The affine relation that indicates that any variable valuation might hold at S is $\mathbf{0} = (0, 0, \cdots, 0)$. Thus, $\mathbf{0}$ holds at S, and the affine relation $\boldsymbol{a}$ holds at T iff $m_1\boldsymbol{a} = m_2\boldsymbol{a} = \cdots = m_r\boldsymbol{a} = \mathbf{0}$. The set of all affine relations that hold at T can be found as the intersection of the null spaces of the matrices $m_1, m_2, \cdots, m_r$.

Extensions to ARA. ARA can also be performed for modular arithmetic [35] to precisely model machine arithmetic (which is modulo 2 to the power of the word size). The weight domain is similar to the one described above.

3.3 Solving for the MOVP Value

There are two algorithms for solving for MOVP values, called *prestar* and *poststar* (by analogy with the algorithms for PDSs). They take as input an automaton that accepts the set of initial configurations. As output, they produce a *weighted automaton*:

Definition 11. *Given a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, a $\mathcal{W}$-**automaton** $\mathcal{A}$ is a $\mathcal{P}$-automaton, where each transition in the automaton is labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in either a forward or backward direction. The automaton is said to accept a configuration $c = \langle p, u \rangle$ with weight $w = \mathcal{A}(c)$ if w is the combine of weights of all accepting paths for u starting from state p in $\mathcal{A}$. We call the automaton a **backward $\mathcal{W}$-automaton** if the weight of a path is read backwards, and a **forward $\mathcal{W}$-automaton** otherwise.*

Let $\mathcal{A}$ be an unweighted automaton and $\mathcal{L}(\mathcal{A})$ be the set of configurations accepted by it. Then, *prestar*$(\mathcal{A})$ produces a forward weighted automaton $\mathcal{A}_{pre^*}$ as output, such that $\mathcal{A}_{pre^*}(c) = \mathrm{MOVP}(\{c\}, \mathcal{L}(\mathcal{A}))$, whereas *poststar*$(\mathcal{A})$ produces a backward weighted automaton $\mathcal{A}_{post^*}$ as output, such that $\mathcal{A}_{post^*}(c) =$

$\mathrm{MOVP}(\mathcal{L}(\mathcal{A}), \{c\})$ [42]. Examples are shown in Fig. 6(b). One thing to note here is how the *poststar* automaton works. The procedure `bar` is analyzed independently of its calling context (i.e., without knowing the exact value of `x`), which generates the transitions between p and p_{n_7}. The calling context of `bar`, which determines the input values to `bar`, is represented by the transitions that leave state p_{n_7}. This is how, for instance, the automaton records that $\mathtt{x} = 3$ and $\mathtt{y} = 3$ at node n_8 when `bar` is called from node n_2.

Using standard automata-theoretic techniques, one can also compute $\mathcal{A}_w(C)$ for (forward or backward) weighted automaton $\mathcal{A}_w$ and a regular set of configurations C, where $\mathcal{A}_w(C) = \bigoplus\{\mathcal{A}_w(c) \mid c \in C\}$. This allows one to solve for the meet-over-all-paths value $\mathrm{MOVP}(S, T)$ for configuration sets S and T by computing either $\mathit{poststar}(S)(T)$ or $\mathit{prestar}(T)(S)$.

We briefly describe how the *prestar* algorithm works for WPDSs. The interested reader is referred to [42] for more details (e.g., the *poststar* algorithm), as well as an efficient implementation of the algorithm. The algorithm takes an unweighted automaton $\mathcal{A}$ as input (i.e., a weighted automaton in which all weights are $\overline{1}$), and adds weighted transitions to it until no more can be added. The addition of transitions is based on the following rule: for a WPDS rule $r = \langle p, \gamma\rangle \hookrightarrow \langle q, \gamma_1 \cdots \gamma_n\rangle$ with weight $f(r)$ and transitions $(q, \gamma_1, q_1), \cdots, (q_{n-1}, \gamma_n, q_n)$ with weights $w_1, \cdots, w_n$, add the transition (p, γ, q_n) to $\mathcal{A}$ with weight $w = f(r) \otimes w_1 \otimes \cdots \otimes w_n$. If this transition already exists with weight w', change the weight to $w \oplus w'$.

This algorithm is based on the intuition that if the automaton accepts configurations c and c' with weights w and w', respectively, and rule r allows the transition $c' \Rightarrow c$, then the automaton needs to accept c' with weight $w' \oplus (f(r) \otimes w)$. Termination follows from the fact that the number of states of the automaton does not increase (hence, the number of transitions is bounded), and the fact that the weight domain satisfies the descending-chain condition (Defn. 4, item 4).

We now provide some intuition into why one needs both forwards and backwards automata. Consider the automata in Fig. 6(c). For the *poststar* automaton, when one follows a path that accepts the configuration $\langle p, n_8\ n_4\rangle$, the transition (p, n_8, q) comes before (q, n_4, acc). However, the former transition describes the transformation inside `bar`, which happens *after* the transformation performed in reaching the call site at n_4 (which is stored on (q, n_4, acc)). Because the transformation for the calling context happens earlier in the program, but its transitions appear later in the automaton, the weights are read backwards. For the *prestar* automaton, the weight on (p, n_4, acc) is the transformation for going from n_4 to n_6, which occurs after the transformation inside `bar`. Thus, it is a forwards automaton.

The following lemma states the complexity for solving *poststar* by the algorithm of Reps et al. [42]. We will assume that the time to perform an $\otimes$ and a $\oplus$ are the same, and use the notation $O_s(.)$ to denote the time bound in terms of semiring operations. The *height* of a weight domain is defined to be the length of the longest descending chain in the domain. For ease of stating a complexity result, we will assume that there is a finite upper bound on the height. Some

weight domains, such as $\mathcal{M}$ in Defn. 8, have no such finite upper bound on the height; however, WPDSs can still be used when the height is unbounded. The absence of infinite descending chains (Defn. 4, item 4) ensures that saturation-based algorithms for computing $post^*$ and pre^* will eventually terminate.

Lemma 1. *[42] Given a WPDS with PDS $\mathcal{P} = (P, \Gamma, \Delta)$, if $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ is a $\mathcal{P}$-automaton that accepts an input set of configurations, poststar produces a backward weighted automaton with at most $|Q| + |\Delta|$ states in time $O_s(|P||\Delta|(|Q_0| + |\Delta|)H + |P||\lambda_0|H)$, where $Q_0 = Q \backslash P$, $\lambda_0 \subseteq \rightarrow$ is the set of all transitions leading from states in Q_0, and H is the height of the weight domain.*

Approximate Analysis. Among the properties imposed by a weight domain, one important property is distributivity (Defn. 4, item 2). This is a common requirement for a precise analysis, which also arises in various *coincidence theorems* for dataflow analysis [22, 47, 26]. Sometimes this requirement is too strict and may be relaxed to monotonicity, i.e., for all $a, b, c \in D$, $a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$. In such cases, the MOVP computation may not be precise, but it will be *safe* under the partial order $\sqsubseteq$.

3.4 Local Variables and Extended Weighted Pushdown Systems

This section discusses an extension of WPDSs that permits abstractions to track the values of local variables [30].

In WPDSs, reachability problems compute the value of a rule sequence by taking an extend of the weights of each of the rules in the sequence; when WPDSs are used for dataflow analysis of a program, rule sequences represent interprocedural paths in the program. To summarize the weights of such paths, we have to maintain information about local variables of all unfinished procedures that appear on the path.

Extended WPDSs (EWPDSs) lift WPDSs to handle local variables in much the same way that Knoop and Steffen lifted conventional dataflow-analysis algorithms to handle local variables [26]: at a call site at which procedure P calls procedure Q, the local variables of P are modeled as if the current incarnations of P's locals are stored in locations that are inaccessible to Q and to procedures transitively called by Q—consequently, the contents of P's locals cannot be affected by the call to Q; we use special merging functions to combine them with the value returned by Q to create the state after Q returns.[3]

For a semiring $\mathcal{S}$ on domain D, a *merging function* is defined as follows:

[3] Note that this model agrees with programming languages like Java, where it is not possible to have pointers to local variables (i.e., pointers into the stack). For languages such as C and C++, where the address-of operator (`&`) allows the address of a local variable to be obtained, if P passes such an address to Q, it is possible for Q (or a procedure transitively called from Q) to affect a local of P by making an indirect assignment through the address.

Conventional interprocedural dataflow-analysis algorithms must also worry about this issue, which is usually dealt with by (i) performing a preliminary analysis to

Definition 12. *A function $g : D \times D \to D$ is a* **merging function** *with respect to a bounded idempotent semiring $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ if it satisfies the following properties.*

1. **Strictness.** *For all $a \in D$, $g(\overline{0}, a) = g(a, \overline{0}) = \overline{0}$.*
2. **Distributivity.** *The function distributes over $\oplus$. For all $a, b, c \in D$,*

$$g(a \oplus b, c) = g(a, c) \oplus g(b, c) \text{ and } g(a, b \oplus c) = g(a, b) \oplus g(a, c)$$

Definition 13. *Let $(\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system; let $\mathcal{G}$ be the set of all merging functions on semiring $\mathcal{S}$, and let Δ_2 denote the set of push rules of $\mathcal{P}$. An* **extended weighted pushdown system** *is a quadruple $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ where $g : \Delta_2 \to \mathcal{G}$ assigns a merging function to each rule in Δ_2.*

Note that a push rule has both a weight and a merging function associated with it. Merging functions are used to fuse the local state of the calling procedure as it existed just before the call with the effects on the global state produced by the called procedure.

As an example, Fig. 2 shows an ICFG and the PDS that represents it. We can perform constant propagation (with uninterpreted expressions) by assigning a weight to each PDS rule. The weight semiring is $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$, where $D = (Env \to Env)$ is the set of all environment transformers, and the semiring operations and constants are defined as follows:

$$\begin{array}{ll} \overline{0} = \lambda e.\top & w_1 \oplus w_2 = \lambda e.(w_1(e) \sqcap w_2(e)) \\ \overline{1} = \lambda e.e & w_1 \otimes w_2 = w_2 \circ w_1 \end{array}$$

The weights for the EWPDS that models the program in Fig. 2 are shown as edge labels. The merging function for the rule $\langle p, n_3 \rangle \hookrightarrow \langle p, e_f\, n_4 \rangle$, which encodes the call at n_3, receives two environment transformers: one that summarizes the effect of the caller from its enter node to the call site (e_{main} to n_3) and one that summarizes the effect of the called procedure (e_f to x_f). The merging function has to produce the transformer that summarizes the effect of the caller from its enter node to the return site (e_{main} to n_4). The merging function is defined as follows:

$$\begin{array}{rcl} g(w_1, w_2) & = & \textbf{if } (w_1 = \overline{0} \text{ or } w_2 = \overline{0}) \textbf{ then } \overline{0} \\ & & \textbf{else } \lambda e.e[a \mapsto w_1(e)(a), y \mapsto (w_1 \otimes w_2)(e)(y)] \end{array}$$

This copies over the value of the local variable a from the call site, and gets the value of y that is returned from the called procedure. Because the merging function has access to the environment transformer just before the call, we do not have to pass the value of local variable a into procedure p. Hence the call stops tracking the value of a using the weight $\lambda e.e[a \mapsto \bot, b \mapsto e(a)]$.

The merging function for the rule $\langle p, n_7 \rangle \hookrightarrow \langle p, e_f\, n_8 \rangle$ is defined similarly.

determine which call sites might have such effects, and (ii) using the results of the preliminary analysis to create sound transformers for the primary analysis. The preliminary analysis is itself an interprocedural dataflow analysis, and (E)WPDSs can be applied to this problem as well. §4 describes how one such preliminary analysis—alias analysis for single-level pointers [32]—can be expressed as a reachability problem in an EWPDS.

Merging Functions for Boolean Programs. In this section, we assume without loss of generality that each procedure has the same number of local variables.

To encode Boolean programs that have local variables, let G be the set of valuations of the global variables and L be the set of valuations of local variables. The actions of program statements and conditions are now binary relations on $G \times L$; thus, the weight domain is a relational weight domain on the set $G \times L$, but with an extra merging function defined on weights. Because different weights can refer to local variables from different procedures, one cannot take relational composition of weights from different procedures. The *project* function is used to change the scope of a weight. It existentially quantifies out the current transformation on local variables and replaces it with an identity relation. Formally, it can be defined as follows:

$$\mathit{project}(w) = \{(g_1, l_1, g_2, l_1) \mid (g_1, l_1, g_2, l_2) \in w\}.$$

Once the summary of a procedure is calculated as a weight w involving local variables of the procedure, the *project* function is applied to it, and the result $\mathit{project}(w)$ is passed to the callers of that procedure. This makes sure that local variables of one procedure do not interfere with those of another procedure. Thus, merging functions for Boolean programs all have the form

$$g(a, b) = a \otimes \mathit{project}(b).$$

For encoding Boolean programs with other abstractions, such as finding the shortest trace, one can use the relational weight domain on $(G \times L, \mathcal{S})$, where $\mathcal{S}$ is a weight domain such as the minpath semiring (transparent to the presence or absence of local variables). The *project* function on weights from this domain can be defined as follows:

$$\mathit{project}(w) = \lambda(g_1, l_1, g_2, l_2).\ \begin{array}{l}\text{if } (l_1 \neq l_2) \text{ then } \overline{0}_{\mathcal{S}} \\ \text{else } \bigoplus_{l \in L} w(g_1, l_1, g_2, l)\end{array}$$

Again, the merging functions all have the form $g(a, b) = a \otimes \mathit{project}(b)$.

4 Case Study: May-Aliasing for Single-Level Pointer Programs

In this section, we define an EWPDS to find variable aliasing in programs written in a C-like imperative language that is restricted to single-level pointers (i.e., one cannot have pointers to pointers).[4] This problem was defined and solved in [32], and has been chosen to illustrate the power of having merging functions in EWPDSs. We first discuss some of the results from [32], and then move on to describe an EWPDS that finds aliasing in a program. For this, we need only

[4] For languages in which more than one level of indirection is possible, the algorithm for single-level pointers still provides a safe solution (i.e., an overapproximation) [32].

to describe the weight domain and merging functions, because we already know how to model the control flow of a program as a PDS (Fig. 3).

We say that two access expressions a and b are aliased (written as $\langle a, b\rangle$) at a particular program point n if in *some* program execution they refer to the same memory location when execution reaches n. We limit access expressions to variables and pointer dereferences (written as $*p$ for an address-valued variable p). Given a program, we want to determine an overapproximation of all alias pairs that hold at each program point. This problem is also referred to as *may-aliasing*. In [32], this is computed in two stages. First, *conditional may-aliasing* information is computed, which answers questions of the form: "if all alias pairs in the set $\mathcal{A}$ hold at a program point n_1, does the pair $\langle a, b\rangle$ hold at point n_2?" The second stage then uses this information to build up the final may-aliasing table.

An important property that results from the fact that we only have single-level pointers is that for all program points n_1 and n_2, where n_1 is the enter node of the procedure containing n_2, if the alias pair $\langle a, b\rangle$ holds at n_2 under the assumption that the set $\mathcal{A} = \{A_1, \cdots, A_m\}$ of alias pairs holds at n_1, then either (i) we can prove that $\langle a, b\rangle$ holds at n_2, assuming that no alias pair holds at n_1; or (ii) there exists a k, $1 \leq k \leq m$, such that assuming that just A_k holds at n_1 suffices to prove that $\langle a, b\rangle$ holds at n_2. In other words, we only need to compute conditional may-alias information for each *alias pair* $A_k \in \mathcal{A}$, rather than for each *subset* of $\mathcal{A}$.

We say that the alias pair $\langle a, .\rangle$ holds at program point n if a is aliased to some access expression that is not visible (out of scope) in the procedure containing n. It is not necessary to know the particular invisible access expression to which a is aliased because a procedure will always have the same effect on all alias pairs that contain access expression a and any invisible access expression [32].

For a given program, let V denote the set of all its variables and pointer dereferences. Assume that all variables have different names (local variables can be prefixed by the name of the procedure that contains them) so that there are no name conflicts. The set $\mathcal{AP} = (V \times V) \cup (V \times \{.\}) \cup (\{.\} \times V)$ is the set of all alias pairs. Let $\mathcal{AP}_\perp = \mathcal{AP} \cup \{\perp\}$, where $\perp$ represents the absence of an alias pair.

We now construct a weight domain over the set $D = (\mathcal{AP}_\perp \rightarrow 2^{\mathcal{AP}})$ of all functions w from $\mathcal{AP}_\perp$ to the power set of $\mathcal{AP}$ with the following monotonicity restriction: for all $x \in \mathcal{AP}$, $w(\perp) \subseteq w(x)$. Operations on weights will maintain the invariant that alias relations are symmetric (i.e., if $\langle a, b\rangle$ holds, so does $\langle b, a\rangle$). Each weight $w \in D$ can be efficiently represented as a one-to-many map from $\mathcal{AP}_\perp$ to $\mathcal{AP}$.

An interprocedural path P with weight w means that if we assume $\langle a, b\rangle$ to hold at the beginning of P then all pairs in $w(\langle a, b\rangle)$ hold at the end of path P when the program execution follows P. The special element $\perp$ handles the case when no pair is assumed to hold at the beginning of the path; $w(\perp)$ is the set of all alias pairs that hold at the end of the path without assuming that any pair holds

at the beginning of the path. Thus, a weight represents conditional may-aliasing information, which motivates the monotonicity condition introduced above.

For all $w_1 \neq \overline{0} \neq w_2$, the semiring operations are defined as follows. For $x \in \mathcal{AP}_\perp$,

$$\begin{aligned}(w_1 \oplus w_2)(x) &= w_1(x) \cup w_2(x)\\ (w_1 \otimes w_2)(x) &= w_2(\perp) \cup (\cup_{y \in w_1(x)} w_2(y))\\ \overline{1}(x) &= \begin{cases} \emptyset & \text{if } x = \perp \\ \{x\} & \text{otherwise} \end{cases}\end{aligned}$$

If path P_1 has weight w_1 and path P_2 has weight w_2, then the weight $w_1 \otimes w_2$ summarizes the conditional alias information of the path P_1 followed by P_2. In particular, $(w_1 \otimes w_2)(x)$ consists of the alias pairs that hold from w_2, regardless of the value of w_1, together with the alias pairs that hold from w_2 given $w_1(x)$. When P_1 and P_2 have the same starting and ending points, the weight $w_1 \oplus w_2$ stores conditional aliasing information when the program execution follows P_1 *or* P_2.

(The semiring constant $\overline{0}$ cannot be naturally described in terms of conditional aliasing, but we can add it to D as a special value that satisfies all properties of Defn. 4.)

We now consider how to associate a weight to each pushdown rule in the EWPDS that encodes the program. For a node n that contains a statement of the form $x = y$, where x and y are pointers, the weight associated with each rule of the form $\langle p, n\rangle \hookrightarrow \cdots$ is a map, where for each $x \in \mathcal{AP}_\perp$, the first applicable mapping is followed:

$$\begin{aligned}\langle *y, b\rangle &\mapsto \{\langle *x, b\rangle\}\\ \langle a, *y\rangle &\mapsto \{\langle a, *x\rangle\}\\ \langle *x, b\rangle &\mapsto \emptyset\\ \langle a, *x\rangle &\mapsto \emptyset\\ \langle a, b\rangle &\mapsto \{\langle a, b\rangle\}\\ \perp &\mapsto \{\langle a, a\rangle \mid a \in V\} \cup \{\langle *x, *y\rangle, \langle *y, *x\rangle\}\end{aligned}$$

Roughly speaking, this generates the alias pairs $\langle *x, *y\rangle$ and $\langle *y, *x\rangle$, makes the aliases of $*y$ into aliases of $*x$, and removes the previously existing alias pairs of $*x$ (except $\langle *x, *x\rangle$). To enforce monotonicity on weights, the following closure operation is applied to the map: $cl(w) = \lambda x.(w(x) \cup w(\perp))$. The weights on other rules that represent intraprocedural edges can be defined similarly (see [32]).

For a *push* rule, the weight is determined according to the binding that occurs at the call site; the definition is presented in Fig. 8. All pop rules have the weight $\overline{1}$.

The merging functions associated with push rules reflect the way conditional aliasing information is computed for return nodes in [32]. Consider the push rule $\langle p, call_{foo}\rangle \hookrightarrow \langle p, enter_{bar}\ return_{foo}\rangle$, which is a call to procedure *bar* from *foo*, and suppose that $bind_{call}$ is the weight associated with this rule. For local access expressions l_1, l_2 of *foo* and global access expressions g_1, g_2, the following must hold.

– The alias pair $\langle l_1, l_2\rangle$ holds at $return_{foo}$ only if the pair $\langle l_1, l_2\rangle$ holds at the call node $call_{foo}$.
– The alias pair $\langle g_1, g_2\rangle$ holds at $return_{foo}$ only if the pair holds at $exit_{bar}$.
– The alias pair $\langle g_1, l_1\rangle$ holds at $return_{foo}$ only if $\langle g_1, .\rangle$ holds at $exit_{bar}$ and the invisible variable is l_1. This happens when a pair $\langle o_1, l_1\rangle$ that held at $call_{foo}$ caused $\langle o_2, .\rangle$ to hold at $enter_{bar}$ because of the call bindings ($\langle o_2, .\rangle \in bind_{call}(\langle o_1, l_1\rangle)$) and this pair, in turn, caused $\langle g_1, .\rangle$ to hold at $exit_{bar}$.

$$bind_n(\bot) = \begin{pmatrix} \{\langle *f_i, *f_j\rangle \mid [f_i, a_i], [f_j, a_j], a_i = a_j\} \\ \cup\, \{\langle *f_i, *a_i\rangle \mid [f_i, a_i], visible_p(a_i)\} \\ \cup\, \{\langle *a_i, *f_i\rangle \mid [f_i, a_i], visible_p(a_i)\} \\ \cup\, \{\langle *f_i, .\rangle \mid [f_i, a_i], \neg visible_p(a_i)\} \\ \cup\, \{\langle ., *f_i\rangle \mid [f_i, a_i], \neg visible_p(a_i)\} \end{pmatrix}$$

$$bind_n(\langle a, b\rangle) = \begin{pmatrix} bind_n(\bot) \\ \cup\, \{\langle a, b\rangle \mid visible_p(a), visible_p(b)\} \\ \cup\, \{\langle a, .\rangle \mid visible_p(a), \neg visible_p(b)\} \\ \cup\, \{\langle ., b\rangle \mid \neg visible_p(a), visible_p(b)\} \\ \cup\, \{\langle a, *f_i\rangle \mid visible_p(a), [f_i, a_i], *a_i = b\} \\ \cup\, \{\langle ., *f_i\rangle \mid \neg visible_p(a), [f_i, a_i], *a_i = b\} \\ \cup\, \{\langle *f_i, b\rangle \mid visible_p(b), [f_i, a_i], *a_i = a\} \\ \cup\, \{\langle *f_i, .\rangle \mid \neg visible_p(b), [f_i, a_i], *a_i = a\} \\ \cup\, \{\langle *f_i, *f_j\rangle \mid [f_i, a_i], [f_j, a_j], *a_i = a, *a_j = b\} \end{pmatrix}$$

Fig. 8. A function that models parameter binding for a call at program point n to a procedure named p. For brevity, we write $[f, a]$ to denote the fact that f is a pointer-valued formal parameter bound to actual a. Also, $visible_p(a)$ is *true* if a is visible in procedure p.

To encode these facts as weights for an algorithmic description of the merging functions, we need to define certain weights and operations on them.

– **Projection.** For a set $S \subseteq (V \cup \{.\})$, let w_S be a weight that only preserves alias pairs in $S \times S$: $w_S(\bot) = \emptyset$ and

$$w_S(\langle a, b\rangle) = \begin{cases} \{\langle a, b\rangle\} & \text{if } a, b \in S \\ \emptyset & \text{otherwise} \end{cases}$$

– **Restoration.** For an access expression $v \in V$, let w_S^v be a weight that changes alias pairs when v comes back in scope conditional on the set $S \subseteq (V \cup \{.\})$: $w_S^v(\bot) = \emptyset$ and

$$w_S^v(\langle a, b\rangle) = \begin{cases} \{\langle a, v\rangle\} & \text{if } b = . \text{ and } a \in S \\ \{\langle v, b\rangle\} & \text{if } a = . \text{ and } b \in S \\ \emptyset & \text{otherwise} \end{cases}$$

– **Conditional Extend.** For an alias pair $\langle a, b\rangle$, define $\otimes_{\langle a,b\rangle}$ to be a binary operation on weights that calculates the alias pairs that hold at the end of a path as a result of the fact that $\langle a, b\rangle$ held at a point inside the path. For $x \in \mathcal{AP}_\perp$,

$$(w_1 \otimes_{\langle a,b\rangle} w_2)(x) = \begin{cases} w_2(\langle a, b\rangle) & \text{if } \langle a, b\rangle \in w_1(x) \\ w_2(\perp) & \text{otherwise} \end{cases}$$

We can now define the merging functions. If G is the set of global access expressions of the program, then for a call from a procedure with local access expressions L and binding weight $bind_{call}$ (i.e., the weight on the push rule), the merging function is defined as follows (where L_e denotes $L \cup \{.\}$):

$$g(w_1, w_2) = \mathbf{if}(w_1 = \overline{0} \ or \ w_2 = \overline{0}) \ \mathbf{then} \ \overline{0} \ \mathbf{else} \begin{pmatrix} (w_1 \otimes w_{L_e}) \\ \oplus \ (w_1 \otimes bind_{call} \otimes w_2 \otimes w_G) \\ \oplus \bigoplus_{\langle a,l\rangle \in V \times L_e} ((w_1 \otimes_{\langle a,l\rangle} (bind_{call} \otimes w_2)) \otimes w^l_G) \\ \oplus \bigoplus_{\langle l,a\rangle \in L_e \times V} ((w_1 \otimes_{\langle l,a\rangle} (bind_{call} \otimes w_2)) \otimes w^l_G) \end{pmatrix}$$

The first term in the combine copies over from the call site the pairs for local access expressions. The second term copies over from the called procedure's exit site the pairs for global access expressions. The third and fourth terms, which are combines over all pairs in $V \times L_e$ and $L_e \times V$, respectively, account for global-local access expressions, following the strategy discussed earlier in this section.

After the EWPDS is constructed, we can run an MOVP query with respect to the configuration set $C = \{\langle p, enter_{main}\rangle\}$ (where p is the single control location of the EWPDS), and obtain the may-alias pairs as follows,

$$\textit{may-alias}(n) = \mathrm{MOVP}(C, n\Gamma^*)(\perp).$$

In addition to computing the Landi-Ryder may-alias pairs, we can also answer stack-qualified queries about may-alias relationships. For instance, we can find out the may-alias pairs that hold at n_1 when execution ends in the stack configuration $\langle p, n_1 n_2 \cdots n_k\rangle$. As discussed in §1, such queries allow us to obtain more precise information than what is obtained by merely computing a may-aliasing query for paths that end at n_1 with *any* stack configuration.

5 Recent Developments

5.1 Improvements in Solver Technology

The algorithms given in [46, 41, 42] are based on saturation (and generalize the saturation procedure used for ordinary unweighted PDSs). Lal and Reps achieved substantial speedups over previous algorithms for WPDS reachability problems by using more sophisticated algorithms in the WPDS solver engine [29].

5.2 Analysis of Concurrent Programs

Two studies have used WPDSs to perform analyses of concurrent programs.

Chaki et al. [12] considers the model-checking problem for concurrent C programs with components that communicate via synchronizing actions (where components use data drawn from large-cardinality data domains and possibly-recursive procedure calls). They model such programs using *communicating pushdown systems*, and reduce the reachability problem for this model to deciding the emptiness of the intersection of two context-free languages L_1 and L_2. Because the latter problem is undecidable, their scheme uses counterexample-guided abstraction refinement of communicating Boolean programs. The technique was implemented as an extension to MAGIC [11], using WPDS++ [24] to perform reachability queries on the models for each component. The system was able to uncover a previously unknown bug in a version of a Windows NT Bluetooth driver.

Lal et al. [31] followed an approach pioneered by Qadeer and Rehof [38], who showed that analysis of concurrent recursive programs is decidable, for a finite-state abstraction of program data, when one limits the amount of concurrency by bounding the number of *context switches*. (A context switch is defined as the transfer of control from one thread to another.)

Such an approach has proven to be useful for program analysis because many bugs can be found in a few context switches [39, 38, 36]. Note that a *context-bounded analysis* (CBA) does not impose any bound on the execution length between context switches. Thus, even with a context-switch bound, the analysis still has to consider the possibility that the next switch takes place in any one of the (possibly infinite) states that may be reached after a context switch. Because of this, CBA still considers many concurrent behaviors [36].

Qadeer and Rehof [38] showed that CBA is decidable for recursive programs under a finite-state abstraction of program data. Lal et al. use WPDSs to generalize the Qadeer-Rehof result to a family of infinite-state abstractions (and also provide a new symbolic algorithm for the finite case). The insight behind the approach is to construct a *weighted transducer* to summarize the execution of a WPDS: the WPDS can go from configuration c_1 to configuration c_2 if and only if the pair (c_1, c_2) is in the language of the transducer. These transducers are composed to solve CBA.

5.3 Polyhedral Analysis

Recently, Denis Gopan in his Ph.D. thesis [19] presented a way to perform numeric program analysis with WPDSs using the polyhedral abstract domain [16]. One of the challenges that he faced was that the polyhedral domain has infinite descending chains, and hence widening techniques are required [13].

Widening is implemented using a weight wrapper that supports the normal weight interface extended with a few extra methods. Two types of weights are used: "regular weights" and "widening weights". Regular weights behave just like ordinary weights; widening weights are placed on WPDS rules where widening

must occur (e.g., rules that correspond to backedges in the ICFG). In particular, if a widening weight b is used in a combine operation by the WPDS saturation procedure, the normal operation $a \oplus b$ is replaced by $a \bigtriangledown (a \oplus b)$, (where $\bigtriangledown$ is the standard widening operator).

References

1. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, August 2007. Tech. Rep. 1603.
2. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
3. G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Computer Aided Verif.*, 2005.
4. T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Spin Workshop*, volume 1885 of *Lec. Notes in Comp. Sci.*, pages 113–130, 2000.
5. T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Prog. Analysis for Softw. Tools and Eng.*, pages 97–103, June 2001.
6. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, volume 1243 of *Lec. Notes in Comp. Sci.*, pages 135–150. Springer-Verlag, 1997.
7. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.*, pages 62–73, 2003.
8. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(6):677–691, August 1986.
9. J.R. Büchi. *Finite Automata, their Algebras and Grammars*. Springer-Verlag, 1988. D. Siefkes (ed.).
10. O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. CONCUR*, volume 630 of *Lec. Notes in Comp. Sci.*, pages 123–137. Springer-Verlag, 1992.
11. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Int. Conf. on Softw. Eng.*, 2003.
12. S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algs. for the Construct. and Anal. of Syst.*, 2006.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Princ. of Prog. Lang.*, pages 238–252, 1977.
14. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.
15. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Princ. of Prog. Lang.*, pages 269–282, 1979.
16. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Princ. of Prog. Lang.*, pages 84–96, 1978.

17. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verif.*, volume 1855 of *Lec. Notes in Comp. Sci.*, pages 232–247, July 2000.
18. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
19. D. Gopan. *Numeric program analysis techniques with applications to array analysis and library summarization*. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, August 2007. Tech. Rep. 1602.
20. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verif.*, volume 1254 of *Lec. Notes in Comp. Sci.*, pages 72–83, 1997.
21. S. Gulwani and G.C. Necula. Precise interprocedural analysis using random interpretation. In *Princ. of Prog. Lang.*, 2005.
22. J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–318, 1977.
23. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6:133–151, 1976.
24. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.
25. G.A. Kildall. A unified approach to global program optimization. In *Princ. of Prog. Lang.*, pages 194–206, 1973.
26. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Comp. Construct.*, pages 125–140, 1992.
27. J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symp.*, 2005.
28. A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *European Symp. on Programming*, 2006.
29. A. Lal and T. Reps. Improving pushdown system model checking. In *Computer Aided Verif.*, 2006.
30. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.
31. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. Tech. Rep. TR-1598, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, July 2007.
32. W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *Princ. of Prog. Lang.*, pages 93–103, January 1991.
33. F. Martin. PAG – An efficient program analyzer generator. *Softw. Tools for Tech. Transfer*, 1998.
34. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Princ. of Prog. Lang.*, 2004.
35. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.
36. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Prog. Lang. Design and Impl.*, 2007.
37. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
38. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Tools and Algs. for the Construct. and Anal. of Syst.*, 2005.
39. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Prog. Lang. Design and Impl.*, 2004.
40. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Princ. of Prog. Lang.*, pages 49–61, 1995.

41. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Static Analysis Symp.*, pages 189–213, 2003.
42. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.*, 58(1–2):206–263, October 2005.
43. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.
44. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
45. S. Schwoon. WPDS: A library for weighted pushdown systems, 2003. http://www.fmi.uni-stuttgart.de/szs/tools/wpds/.
46. S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Comp. Sec. Found. Workshop*, 2003.
47. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
48. J. Whaley, D. Avots, M. Carbin, and M.S. Lam. Using Datalog with Binary Decision Diagrams for program analysis. In *Asian Symp. on Prog. Lang. and Systems*, 2005.