Program Analyses using Newton's Method*

(Invited Paper)

Thomas Reps[1,2] **

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. Esparza et al. generalized Newton's method—a numerical-analysis algorithm for finding roots of real-valued functions—to a method for finding fixed-points of systems of equations over semirings. Their method provides a new way to solve interprocedural dataflow-analysis problems. As in its real-valued counterpart, each iteration of their method solves a simpler "linearized" problem.

Because essentially all fast iterative numerical methods are forms of Newton's method, this advance is exciting because it may provide the key to creating faster program-analysis algorithms. However, there is an important difference between the dataflow-analysis and numerical-analysis contexts: when Newton's method is used in numerical problems, commutativity of multiplication is relied on to rearrange an expression of the form "$a*Y*b+c*Y*d$" into "$(a*b+c*d)*Y$." Equations with such expressions correspond to path problems described by regular languages. In contrast, when Newton's method is used for interprocedural dataflow analysis, the "multiplication" operation involves function composition, and hence is non-commutative: "$a*Y*b+c*Y*d$" cannot be rearranged into "$(a*b+c*d)*Y$." Equations with the former expressions correspond to path problems described by linear context-free languages (LCFLs).

The invited talk that this paper accompanies presented a method that we developed in 2015 for solving the LCFL sub-problems produced during successive rounds of Newton's method. It uses some algebraic slight-of-hand to turn a class of LCFL path problems into regular-language path problems. This result is surprising because a reasonable sanity check—formal-language theory—suggests that it should be impossible: after all, the LCFL languages are a strict superset of the regular languages.

The talk summarized several concepts and prior results on which that result is based. The method described applies to predicate abstraction, on which most of today's software model checkers rely, as well as to other abstract domains used in program analysis.

1 Introduction

Static program analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually run-

* Portions of this work are excerpted from [12].

** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.

ning the program on specific inputs. Instead, static-analysis techniques explore a program's behavior for all possible inputs and all possible states that the program can reach. To make this approach feasible, the program is "run in the aggregate"—i.e., on descriptors that represent collections of many states.

This paper briefly reviews the conventional approach to interprocedural dataflow analysis, and then summarizes a line of work from the last ten years in which Newton's method—a numerical-analysis algorithm for finding roots of real-valued functions—has been generalized so that it can be used as a method for finding solutions to the systems of equations that arise in interprocedural dataflow-analysis problems.

2 Interprocedural Dataflow Analysis

Example 1. Consider the following program scheme, where X_1 represents the main procedure, X_2 represents a subroutine, and s_a, s_b, s_c, and s_d represent four program statements:

```
X1() {                    X2() {
    sa;                       if (⋆)  sd
    X2()                      else {
}                                 sb; X2(); X2(); sc
                              }
                          }
```

Suppose that we have a domain of functions whose elements correspond to some abstraction of the state-transformers of the programming language. (An example of such a domain is the domain of relations used to formulate an analysis based on predicate abstraction [6].) Let a, b, c, and d denote the elements that abstract the actions of statements s_a, s_b, s_c, and s_d, respectively. The (abstract) actions of procedures X_1 and X_2 can be expressed as the following set of recursive equations:

$$X_1 = a \otimes X_2 \qquad X_2 = d \oplus b \otimes X_2 \otimes X_2 \otimes c, \tag{1}$$

where $\oplus$ (*combine*) denotes the operation used to combine (or "join") information that flows along different paths to some variable, and $\otimes$ (*extend*) is an abstraction of (the reversal of) function composition. Such an equation system can also be viewed as a representation of a program's interprocedural control-flow graph (CFG). (See Fig. 1(a).)

Each unknown in an equation system represents an abstract transformer that serves as a *procedure summary*, in the sense of Sharir & Pnueli [13] and Reps et al. [11]. A summary for procedure X_i overapproximates the behavior of X_i, including all procedures called transitively from X_i. Once procedure summaries have been obtained, one can use them to analyze each procedure X_j to obtain an abstract value $V_{j,p}$ for each program point p in X_j [13]. $V_{j,p}$ represents a superset of the states that can arise at p. To find potential bugs, for instance, one needs to determine if any bad states can arise at p, which can be done by

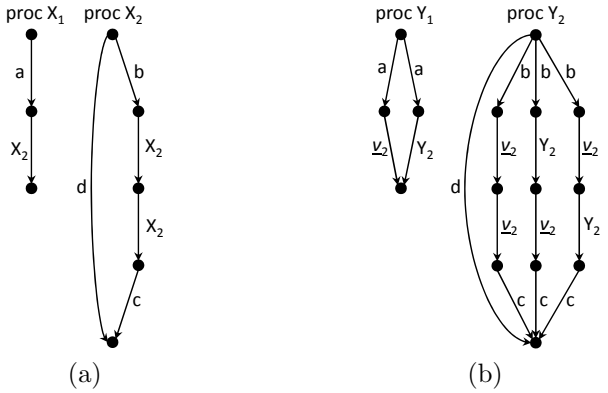

Fig. 1. (a) Graphical depiction of the equation system given in Eqn. (1) as an interprocedural control-flow graph. The three edges labeled "X_2" represent calls to procedure X_2. (b) Linearized equation system over $\{Y_1, Y_2\}$ obtained from Eqn. (1) via Eqn. (6); see Ex. 3.

checking whether any bad states are contained within the meaning of $V_{j,p}$—in abstract-interpretation terminology, by checking whether $(\gamma(V_{j,p}) \cap \text{Bad}) \neq \emptyset$. □

A key goal of an interprocedural analyzer is to obtain a procedure summary for each procedure of the program. The reason is that with a summary function in hand for each procedure, one can reduce the problem of solving an interprocedural dataflow-analysis problem to that of solving a collection of intraprocedural dataflow-analysis problems.[3]

Problem Statement:
Given a set of possibly recursive procedures $\mathcal{P} = \{P_i\}$, and an abstract semantics, i.e.,
- a transformer $f[m_i, n_i]$ on each edge (m_i, n_i) in the control-flow graph of each procedure P_i,
- extend ($\otimes$) and combine ($\oplus$) operators,

find a procedure summary $\varphi[s_i, x_i]$ for each $P_i \in \mathcal{P}$, where s_i and x_i denote the start node and exit node of P_i, respectively.

The conventional approach to computing procedure-summary functions is to work with the set of variables

$$\Phi \stackrel{\text{def}}{=} \{\varphi[s_i, n_i] \mid 1 \leq i \leq |\mathcal{P}|,\ s_i \text{ the start node of } P_i, \text{ and } n_i \text{ a node of } P_i\},$$

and set up the following equation system over Φ:

[3] See [2, §5.1] for an interprocedural dataflow-analysis method that uses a somewhat similar approach.

$$
\begin{aligned}
\varphi[s,s] &= Id && \text{for all } s \in StartNodes \\
\varphi[s_i,n_i] &= \bigoplus_{(m_j,n_i)\in Edges_i} \varphi[s_i,m_j] \otimes f[m_j,n_i] && \text{for all } \begin{cases} s_i \in StartNodes \\ \wedge\ n_i \notin StartNodes \\ \wedge\ s_i, m_j, n_i \in Nodes_i \end{cases} \\
\varphi[s_i,r_i] &= \varphi[s_i,c_i] \otimes In_{c_i,s'} \otimes \varphi[s',x'] \otimes Out_{x',r_i} && \text{for all } \begin{cases} (c_i,r_i) \in CallSites \\ \wedge\ (c_i,s') \in Calls \\ \wedge\ (s',x') \in StartExitPairs \\ \wedge\ s_i, c_i, r_i \in Nodes_i \end{cases}
\end{aligned}
\tag{2}
$$

Eqn. (2) is then solved using a successive-approximation method (i.e., Kleene evaluation or chaotic iteration). Essentially this approach was proposed independently and contemporaneously by Cousot & Cousot [3] and Sharir & Pnueli [13].[4] Note that by solving Eqn. (2), one obtains values for the set of functions $\Phi = \{\varphi[s_i,n_i]\}$, which contains more than just the set of summary functions $\{\varphi[s_i,x_i] \mid (s,x) \in StartExitPairs\}$ (which were referred to as the variables $\{X_i\}$ in Ex. 1).

It is useful to formalize these concepts by introducing the notion of a semiring.

Definition 1. *A* ***semiring*** $\mathcal{S} = (D, \oplus, \otimes, \underline{0}, \underline{1})$ *consists of a set of* ***elements*** D *equipped with two binary operations:* ***combine*** *($\oplus$) and* ***extend*** *($\otimes$). $\oplus$ and $\otimes$ are associative, and have identity elements $\underline{0}$ and $\underline{1}$, respectively. $\oplus$ is commutative, and $\otimes$ distributes over $\oplus$. (A semiring is sometimes called a* ***weight domain****, in which case elements are called* ***weights****.)*

Definition 2. *If A is a finite set, the* relational weight domain *on A is defined as $(\mathcal{P}(A \times A), \cup, ;, \emptyset, id)$. A weight $R \subseteq A \times A$ is a binary relation on A,*[5] *$\oplus$ is union ($\cup$), $\otimes$ is relational composition ($;$), $\underline{0}$ is the empty relation, and $\underline{1}$ is the identity relation on A.*

Example 2. Defn. 2 gives us a way to formalize each predicate-abstraction domain as a semiring. A Boolean program is a program whose only datatype is Boolean. A Boolean program P can be used as an abstraction of a real-world program [1] via predicate abstraction. For each predicate p, there is a variable $v_p \in Var$ in the Boolean program, which holds the value of p in states of the program being modeled. A state of the Boolean program is an assignment in $Var \to$ Bool.

By instantiating A in Defn. 2 to be the set of assignments $Var \to$ Bool, we obtain a semiring whose values can encode the state-transformers of P: the semiring value associated with an assignment statement or assume statement `st` of P is the binary relation on A that represents the effect of `st` on the state of P. □

In this paper, the focus is on semirings in which $\oplus$ is *idempotent* (i.e., for all $a \in D$, $a \oplus a = a$). In an idempotent semiring, the order on elements is defined

[4] Extensions for handling local variables are given by Knoop & Steffen [8], Müller-Olm & Seidl [10], and Lal et al. [9].

[5] A weight can also be thought of as a Boolean matrix with dimensions $|A| \times |A|$.

by $a \sqsubseteq b$ iff $a \oplus b = b$. (Idempotence would be expected in the context of dataflow analysis because an idempotent semiring is a join semilattice $(D, \oplus)$ in which the join operation is $\oplus$.)

A semiring is *commutative* if for all $a, b \in D$, $a \otimes b = b \otimes a$. In dataflow analysis, we typically work with *non-commutative* semirings: the $\otimes$ operation used in Eqn. (2) is an abstraction of (the reversal of) function composition, and hence, in general, is not commutative.

3 Newtonian Program Analysis

Newtonian Program Analysis (NPA) provides an alternative to Kleene iteration or chaotic iteration for solving an equation system such as Eqn. (2). The first step in the story of its development was a method developed for analyzing properties of Recursive Markov Chains (RMCs), which are a modeling formalism for probabilistic programs with possibly recursive procedures. To determine termination probabilities for vertices in a model, Etessami and Yannakakis [5] generate a set of equations that are, in general, non-linear. Answers can be irrational numbers, so one cannot hope to compute them exactly. Instead, their goal is to approximate the probabilities, or to answer decision questions (such as whether the probability is greater than or equal to a specific rational value). They developed an algorithm that uses a multivariate Newton's method for approximating probabilities. Moreover, they showed that, in the limit, when their method is started from the zero vector, it always converges to the correct answer. (In contrast, for general nonlinear polynomial equations over the reals, Newton's method is not guaranteed to converge.)

Etessami and Yannakakis worked with the probability semiring (namely, $(\mathbb{R}_0 \cup \{+\infty\}, +, \times, 0, 1)$), which has numeric values and a commutative $\otimes$ operation. Their work inspired Esparza et al. [4] to investigate whether a generalization of the approach could be applied to other kinds of program analyses—in particular, when control-flow graph edges are labeled with values from a semiring other than the probability semiring. The attempt was successful, and they developed a method for finding the least fixed-point of a system of equations over a semiring, which works for both commutative and *non-commutative* semirings, and does not require that the values be totally ordered. The fact that it applies when $\otimes$ is not commutative makes the work applicable to the problem of finding procedure-summary functions.

In general, let $\mathcal{S} = (D, \oplus, \otimes, \underline{0}, \underline{1})$ be a semiring and $a_1, \ldots, a_{k+1} \in D$ be semiring elements. Let $\mathcal{X}$ be a finite set of variables $X_1, \ldots, X_k$. A *monomial* is a finite expression $a_1 X_1 a_2 \ldots a_k X_k a_{k+1}$, where $k \geq 0$. Monomials of the form $X_1 a_2$, $a_1 X_1$, and $a_1 X_1 a_2$ are *left-linear*, *right-linear*, and *linear*, respectively. (A monomial that consists of a single semiring constant a_1 is considered to be left-linear, right-linear, and linear.) A *polynomial* is a finite expression of the form $m_1 \oplus \ldots \oplus m_p$, where $p \geq 1$ and $m_1, \ldots, m_p$ are monomials. A polynomial is linear if all of its monomials are linear. A system of polynomial equations has

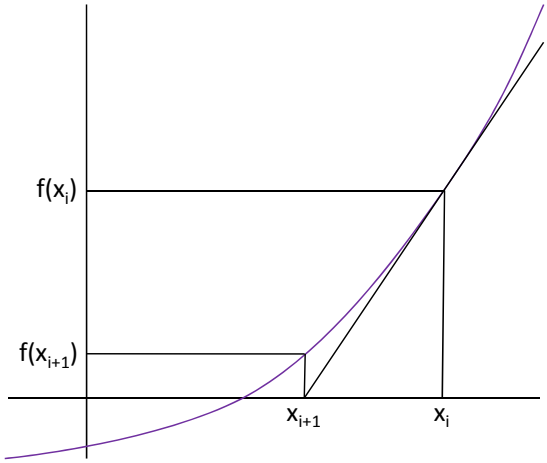

Fig. 2. The principle behind Newton's method for finding roots of real-valued functions.

the form

$$\begin{aligned} X_1 &= f_1(X_1, \ldots, X_n) \\ &\cdots \\ X_n &= f_n(X_1, \ldots, X_n), \end{aligned}$$

or, equivalently, $\overrightarrow{X} = \overrightarrow{f}(\overrightarrow{X})$, where $\overrightarrow{X} = \langle X_1, \ldots, X_n \rangle$ and $\overrightarrow{f} = \lambda \overrightarrow{X}.\langle f_1(\overrightarrow{X}), \ldots, f_n(\overrightarrow{X}) \rangle$. For instance, for Eqn. (1),

$$\overrightarrow{f} \stackrel{\text{def}}{=} \lambda \overrightarrow{X}.\langle a \otimes X_2, d \oplus b \otimes X_2 \otimes X_2 \otimes c \rangle.$$

In numerical problems, the workhorse for successive-approximation algorithms is Newton's method. Fig. 2 illustrates how Newton's method can (sometimes) help identify where a root of a function lies. (Newton's method is not guaranteed to converge to a root.) The general principle is to create a *linear model* of the function—in this case the tangent line—and solve the problem for the linear model to obtain the next approximation to the root of the original function.

Compared to the numerical setting, Esparza et al. had two issues that they needed to finesse:

1. With numerical functions, the linear model is defined using derivatives, which are defined in terms of limits. We have no analogue of a limit in a semiring.
2. Newton's method is for root-finding (i.e., find x such that $f(x) = 0$), whereas in program analysis we are interested in finding a fixed-point (i.e., find x such that $f(x) = x$). Although one can easily convert a fixed-point problem into a root-finding problem—find x such that $f(x) - x = 0$—this approach creates a new problem because there is no analogue of a subtraction operation in a semiring.

Kleene iteration is the well-known technique for finding the least fixed-point of $\overrightarrow{X} = \overrightarrow{f}(\overrightarrow{X})$ via the successive-approximation method

$$\boxed{\begin{array}{rcl} \overrightarrow{\kappa}^{(0)} & = & \overrightarrow{\bot} \\ \overrightarrow{\kappa}^{(i+i)} & = & \overrightarrow{f}(\overrightarrow{\kappa}^{(i)}) \end{array}} \tag{3}$$

The NPA method of Esparza et al. [4] provides an alternative method for finding the least fixed-point of $\overrightarrow{X} = \overrightarrow{f}(\overrightarrow{X})$. NPA is also a successive-approximation method, but uses the following iterative scheme:[6]

$$\boxed{\begin{array}{rcl} \overrightarrow{\nu}^{(0)} & = & \overrightarrow{\bot} \\ \overrightarrow{\nu}^{(i+1)} & = & \overrightarrow{f}(\overrightarrow{\nu}^{(i)}) \sqcup \text{LinearCorrectionTerm}(\overrightarrow{f}, \overrightarrow{\nu}^{(i)}) \end{array}} \tag{4}$$

where LinearCorrectionTerm$(\overrightarrow{f}, \overrightarrow{\nu}^{(i)})$ is a correction term—a function of $\overrightarrow{f}$ and the current approximation $\overrightarrow{\nu}^{(i)}$—that nudges the next approximation $\overrightarrow{\nu}^{(i+1)}$ in the right direction at each step. In essence, the insight behind the work of Esparza et al. is that the high-level principle of Newton's method, namely,

> repeatedly, create a linear model of the function and use it to find a better approximation of the solution

can be applied to programs, too. The sense in which the correction term in Eqn. (4) is "linear" is what makes it proper to say that Eqn. (4) is a form of Newton's method.

More precisely, NPA solves the following sequence of problems for $\overrightarrow{\nu}$:

$$\boxed{\begin{array}{rcl} \overrightarrow{\nu}^{(0)} & = & \overrightarrow{f}(\overrightarrow{0}) \\ \overrightarrow{\nu}^{(i+1)} & = & \overrightarrow{Y}^{(i)} \end{array}} \tag{5}$$

where $\overrightarrow{Y}^{(i)}$ is the value of $\overrightarrow{Y}$ in the least solution of

$$\boxed{\overrightarrow{Y} = \overrightarrow{f}(\overrightarrow{\nu}^{(i)}) \oplus \mathcal{D}\overrightarrow{f}|_{\overrightarrow{\nu}^{(i)}}(\overrightarrow{Y})} \tag{6}$$

and $\mathcal{D}\overrightarrow{f}|_{\overrightarrow{\nu}^{(i)}}(\overrightarrow{Y})$ is the *multivariate differential* of $\overrightarrow{f}$ at $\overrightarrow{\nu}^{(i)}$, defined below (see Defn. 3). Eqns. (5) and (6) resemble Kleene iteration, except that on each iteration $\overrightarrow{f}(\overrightarrow{\nu}^{(i)})$ is "corrected" by the amount $\mathcal{D}\overrightarrow{f}|_{\overrightarrow{\nu}^{(i)}}(\overrightarrow{Y})$.

There is a close analogy between NPA and the use of Newton's method in numerical analysis to solve a system of polynomial equations $\overrightarrow{f}(\overrightarrow{X}) = \overrightarrow{0}$. In both cases, one creates a linear approximation of $\overrightarrow{f}$ around the point $(\overrightarrow{\nu}^{(i)}, \overrightarrow{f}(\overrightarrow{\nu}^{(i)}))$, and then uses the solution of the linear system in the next approximation of $\overrightarrow{X}$. The sequence $\overrightarrow{\nu}^{(0)}, \overrightarrow{\nu}^{(1)}, \ldots, \overrightarrow{\nu}^{(i)}, \ldots$ is called the *Newton sequence* for

[6] For reasons that are immaterial to this discussion, Esparza et al. start the iteration via $\overrightarrow{\nu}^{(0)} = \overrightarrow{f}(\overrightarrow{\bot})$, rather than $\overrightarrow{\nu}^{(0)} = \overrightarrow{\bot}$. Our goal here is to bring out the essential similarities between Eqns. (3) and (4).

$\overrightarrow{X} = \overrightarrow{f}(\overrightarrow{X})$. The process of solving Eqns. (5) and (6) for $\overrightarrow{\nu}^{(i+1)}$, given $\overrightarrow{\nu}^{(i)}$, is called a *Newton step* or one *Newton round*.

For polynomial equations over a semiring, the linear approximation of $\overrightarrow{f}$ is created by the transformation given in Defn. 3. It converts a system of equations with polynomial right-hand sides into a new equation system in which each equation's right-hand side is linear.

Definition 3. *[4] Let $f_i(\overrightarrow{X})$ be a component function of $\overrightarrow{f}(\overrightarrow{X})$. The **differential** of $f_i(\overrightarrow{X})$ with respect to X_j at $\underline{\overrightarrow{\nu}}$, denoted by $\mathcal{D}_{X_j} f_i|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y})$, is defined as follows:*

$$\boxed{\mathcal{D}_{X_j} f_i|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \stackrel{\text{def}}{=} \begin{cases} \underline{0} & \text{if } f_i = s \in \mathcal{S} \\ \underline{0} & \text{if } f_i = X_k \text{ and } k \neq j \\ Y_j & \text{if } f_i = X_j \\ \bigoplus\limits_{k \in K} \mathcal{D}_{X_j} g_k|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) & \text{if } f_i = \bigoplus\limits_{k \in K} g_k \\ \left(\begin{array}{c} \mathcal{D}_{X_j} g|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \otimes h(\underline{\overrightarrow{\nu}}) \\ \oplus\, g(\underline{\overrightarrow{\nu}}) \otimes \mathcal{D}_{X_j} h|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \end{array}\right) & \text{if } f_i = g \otimes h \end{cases}} \tag{7}$$

*where $K \subseteq \mathbb{N}$ is some finite or infinite index set. Let $\overrightarrow{f}$ be a multivariate polynomial function defined by $\overrightarrow{f} \stackrel{\text{def}}{=} \lambda \overrightarrow{X}.\langle f_1(\overrightarrow{X}), \ldots, f_n(\overrightarrow{X})\rangle$. The **multivariate differential** of $\overrightarrow{f}$ at $\underline{\overrightarrow{\nu}}$, denoted by $\mathcal{D}\overrightarrow{f}|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y})$, is defined as follows:*

$$\boxed{\mathcal{D}\overrightarrow{f}|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \stackrel{\text{def}}{=} \left\langle \begin{array}{c} \mathcal{D}_{X_1} f_1|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \oplus \ldots \oplus \mathcal{D}_{X_n} f_1|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}), \\ \vdots \\ \mathcal{D}_{X_1} f_n|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \oplus \ldots \oplus \mathcal{D}_{X_n} f_n|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \end{array} \right\rangle}$$

$\mathcal{D} f_i|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \stackrel{\text{def}}{=} \bigoplus\limits_{k=1}^{n} \mathcal{D}_{X_k} f_i|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y})$ *denotes the i^{th} component of $\mathcal{D}\overrightarrow{f}|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y})$.*

The fourth case in Eqn. (7) generalizes the differential of a binary combine, i.e.,

$$\mathcal{D}_{X_j} g_1|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \oplus \mathcal{D}_{X_j} g_2|_{\underline{\overrightarrow{\nu}}}(\overrightarrow{Y}) \qquad \text{if } f_i = g_1 \oplus g_2,$$

to infinite combines. Note how the fifth case, for "$g \otimes h$", resembles the product rule from differential calculus

$$\frac{d}{dx}(g * h) = \frac{dg}{dx} * h + g * \frac{dg}{dx},$$

and in particular the differential form of the product rule:

$$d(g * h) = dg * h + g * dh.$$

The multivariate differential defined in Defn. 3 is how Esparza et al. addressed the issue raised in item (1) above: the multivariate differential is a *formal operator*

on a polynomial expression over a semiring, and does not involve any notion of "the limit as ΔX approaches 0." Here Esparza et al. [4] were inspired by a formal differentiation operation defined by Hopkins & Kozen [7] for *commutative* Kleene algebra. Esparza et al. generalized that notion to one for creating a formal differential for an equation system defined over a *non-commutative* semiring.

Example 3. For Eqn. (1), the multivariate differential of $\overrightarrow{f}$ at the value $\overrightarrow{\underline{\nu}} = \langle \underline{\nu}_1, \underline{\nu}_2 \rangle$ is

$$\begin{aligned}
\mathcal{D}\overrightarrow{f}|_{(\underline{\nu}_1,\underline{\nu}_2)}(\overrightarrow{Y}) &= \left\langle \begin{array}{l} \mathcal{D}_{X_1} f_1|_{(\underline{\nu}_1,\underline{\nu}_2)}(\overrightarrow{Y}) \oplus \mathcal{D}_{X_2} f_1|_{(\underline{\nu}_1,\underline{\nu}_2)}(\overrightarrow{Y}), \\ \mathcal{D}_{X_1} f_2|_{(\underline{\nu}_1,\underline{\nu}_2)}(\overrightarrow{Y}) \oplus \mathcal{D}_{X_2} f_2|_{(\underline{\nu}_1,\underline{\nu}_2)}(\overrightarrow{Y}) \end{array} \right\rangle \\
&= \left\langle \underline{0} \oplus a \otimes Y_2, \underline{0} \oplus \begin{pmatrix} \underline{0} \\ \oplus\ b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \right\rangle \\
&= \left\langle a \otimes Y_2, \begin{pmatrix} b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \right\rangle \qquad (8)
\end{aligned}$$

From Eqn. (6), we then obtain the following linearized system of equations, which is also depicted graphically in Fig. 1(b):

$$\langle Y_1, Y_2 \rangle \ = \ \left\langle \begin{pmatrix} a \otimes \underline{\nu}_2 \\ \oplus\ a \otimes Y_2 \end{pmatrix}, \begin{pmatrix} d \\ \oplus\ b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \right\rangle \qquad (9)$$

On the $i+1^{st}$ Newton round, we need to solve Eqn. (9) for $\langle Y_1, Y_2 \rangle$ with $\langle \underline{\nu}_1, \underline{\nu}_2 \rangle$ set to the value $\langle \nu_1^{(i)}, \nu_2^{(i)} \rangle$ obtained on the i^{th} round, and then perform the assignment $\langle \nu_1^{(i+1)}, \nu_2^{(i+1)} \rangle \leftarrow \langle Y_1, Y_2 \rangle$.

NPA can also be thought of as a kind of sampling method for the state space of the program. For instance, in Ex. 1, procedure X_2 has two call-sites. In the corresponding linearized program in Fig. 1(b), each path through Y_2 has at most one call site: the NPA linearizing transformation inserted the value $\underline{\nu}_2$ at various call-sites, and left at most one variable in each summand. In essence, during a given Newton round the analyzer samples the state space of Y_2 by taking the $\oplus$ of various paths through Y_2. Along each such path, the abstract values for the call-sites encountered are held fixed at $\underline{\nu}_2$, except for possibly one call-site on the path, which is explored by visiting (the linearized version of) the called procedure. The abstract values for $\underline{\nu}_1$ and $\underline{\nu}_2$ are updated according to the results of this state-space exploration, and the algorithm proceeds to the next Newton round. □

4 Newtonian Program Analysis via Tensor Product (NPA-TP)

Consider the (recursive) equation for Y_2:

$$Y_2 = \begin{pmatrix} d \\ \oplus\ b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \tag{10}$$

Each monomial in Eqn. (10) is *linear*. In contrast, the equation for X_2 in the original equation system (Eqn. (1)), $X_2 = d \oplus b \otimes X_2 \otimes X_2 \otimes c$, involves a monomial that is *quadratic*. In general, as in the example above, NPA reduces the problem of solving an equation system that involves polynomial right-hand sides to the problem of solving a sequence of equation systems, each of which has only linear right-hand sides.

At first blush, one might think (as the author did at one point) that NPA reduces the problem of solving a polynomial equation system for an *inter*procedural dataflow-analysis problem to a sequence of *intra*procedural dataflow-analysis problems. That would be desirable because there exist fast methods to solve an intraprocedural dataflow-analysis problem. For instance, Tarjan [15] introduced the idea of using regular expressions as a kind of "most-general dataflow-analysis method." Specific dataflow-analysis problems are solved by first solving the *path-expression problem*: a program's CFG is considered to be a finite-state machine in which CFG nodes are states, and each edge is labeled by an alphabet symbol unique to that edge. Tarjan's path-expression method [14] creates for each node n a regular expression R_n whose language, $L(R_n)$, is (exactly) the set of all paths from the CFG's start node to n. The "client" dataflow-analysis problem is then solved by evaluating each regular expression R_n, bottom up, using a suitable interpretation, in which the regular-expression operators $+$, $\cdot$, and $*$—now treated as syntactic operators—are interpreted as some suitable (sound) operations, $\oplus$, $\otimes$, and $*$, respectively, in the analysis domain [15].

Because both the regular languages (Reg) and the linear context-free languages (LCFLs) will play a role in what follows, it is worth recalling a few facts.

- $L(FSM) = L(RegExp) = \text{Reg}$
- For a *left-linear* context-free grammar, e.g., $W ::= Wc \mid Wd \mid \epsilon$, $L(W) \in \text{Reg}$. For instance, $L(W) = (c + d)^* \in \text{Reg}$.
- For the *linear* context-free grammar $W ::= aWb \mid \epsilon$, $L(W) = \{a^i b^i\} \notin \text{Reg}$.
- For the *linear* context-free grammar $W ::= a_1 W b_1 \mid a_2 W b_2 \mid \epsilon$,

$$L(W) = \{\ldots, \underbrace{a_1\, \underbrace{a_2 b_2}\, b_1}, \underbrace{a_2\, \underbrace{a_1 b_1}\, b_2}, \underbrace{a_2\, \underbrace{a_1\, \underbrace{a_2 b_2}\, b_1}\, b_2}, \ldots\} \notin \text{Reg}.$$

  In particular, note the "mirrored symmetry" of each word in $L(W)$, as shown above by the underbraces.

Returning to Eqn. (10), note that the third and fourth monomials each extend Y_2 by nontrivial quantities on both the left and the right. Thus, we are truly working with a linear equation system—*not one that is left-linear or right-linear.* In other words, there is a mismatch:

- Tarjan's method solves a left-linear (or right-linear) system of equations.
- The NPA method of Esparza et al. repeatedly creates a linear system that needs to be solved.
- However, in general, one cannot apply Tarjan's method to the linear systems created by NPA.

One can also consider Eqn. (10) as defining the following linear context-free grammar over the set of nonterminals $\{Y_2\}$ and the set of terminals $\{b, c, d, \underline{\nu}_2\}$:

$$Y_2 ::= d \mid b\ \underline{\nu}_2\ \underline{\nu}_2\ c \mid b\ Y_2\ \underline{\nu}_2\ c \mid b\ \underline{\nu}_2\ Y_2\ c \tag{11}$$

Definition 4. *An equation system over semiring $\mathcal{S}$ is an **LCFL equation system** if each equation has the form*

$$Y_j = c_j \oplus \bigoplus_{i,k} (a_{i,j,k} \otimes Y_i \otimes b_{i,j,k}),$$

where $a_{i,j,k}, b_{i,j,k}, c_j \in \mathcal{S}$.

As mentioned earlier, NPA performs a Kleene-like iteration, during which a linear correction is applied on each round. Defn. 4 allows us to be more precise: the correction value used on each round is the solution to an LCFL equation system. The contribution of NPA-TP to NPA is to address the following problem:

Given an LCFL equation system $\mathcal{L}$, devise an efficient method for finding the least solution of $\mathcal{L}$.

Definition 5. *An LCFL equation system over semiring $\mathcal{S}$ is a **left-linear equation system** if each equation has the form*

$$Z_j = c_j \oplus \bigoplus_{i,k} (Z_i \otimes b_{i,j,k}),$$

where $b_{i,j,k}, c_j \in \mathcal{S}$.

In contrast to a general LCFL equation system (Defn. 4), with a left-linear equation system one can always collect coefficients for a given Z_i—i.e., $d_{i,j} = \bigoplus_k b_{i,j,k}$—so that equations can always be put in a form in which Z_j has a single dependence on each Z_i:

$$Z_j = c_j \oplus \bigoplus_{i} (Z_i \otimes d_{i,j}),$$

where $c_j, d_{i,j} \in \mathcal{S}$.

A left-linear equation system corresponds to a left-linear grammar, and hence a regular language. The fact that Tarjan's path-expression method provides a

fast method for solving left-linear equation systems led us to pose the following question:

Is it possible to "regularize" the LCFL equation system $\mathcal{L}$ that arises on each Newton round—i.e., transform $\mathcal{L}$ into a left-linear equation system $\mathcal{L}_{\text{Reg}}$?

If the extend ($\otimes$) operation of the semiring is commutative, it is trivial to turn an LCFL equation system into a left-linear equation system. However, in dataflow-analysis problems, we rarely have a commutative extend operation; thus, our goal was to find a way to regularize a *non-commutative* LCFL equation system.

On the face of it, this line of attack seems unlikely to pan out; after all, Eqn. (11) resembles the *linear* grammar $\overrightarrow{Y} ::= a_1 \overrightarrow{Y} b_1 \mid a_2 \overrightarrow{Y} b_2 \mid \epsilon$, whose simpler cousin $\overrightarrow{Y} ::= a \overrightarrow{Y} b \mid \epsilon$, which is also a linear grammar, generates the language $L(\overrightarrow{Y}) = \{a^i b^i \mid i \in \mathbb{N}\}$—the canonical example of an LCFL that is not regular! For the linear context-free grammar $\overrightarrow{Y} ::= a_1 \overrightarrow{Y} b_1 \mid a_2 \overrightarrow{Y} b_2 \mid \epsilon$,

$$L(\overrightarrow{Y}) = \{\ldots, a_1 \underbrace{a_2 b_2}\, b_1, a_2 \underbrace{a_1 b_1}\, b_2, a_2 \underbrace{a_1 \underbrace{a_2 b_2}\, b_1}\, b_2, \ldots\} \notin \text{Reg}, \tag{12}$$

In particular, note the "mirrored symmetry" of each word in $L(\overrightarrow{Y})$, as shown above by the underbraces. Any solution to the problem of regularizing a non-commutative LCFL equation system has to accommodate such mirrored correlation patterns.

The challenge is to devise a way to accumulate matching quantities on both the left and right sides, whereas in a regular language, we can only accumulate values on one side. What we contributed in [12] is a way, under certain conditions, to convert each LCFL equation system into a left-linear (and hence regular-language) equation system. The result is surprising because a reasonable sanity check—formal-language theory—suggests that it should be impossible: the LCFLs are strictly more expressive than the regular languages.

The secret is that we are not working with words: the combine ($\oplus$) and extend ($\otimes$) operators of the semiring do not denote alternation and concatenation, as in formal-language theory; on the contrary, $\oplus$ and $\otimes$ are interpreted operators. In [12], we used some algebraic slight-of-hand to turn a class of LCFL equation systems into left-linear equation systems. To accomplish such a transformation, we require the semiring to support a few additional operations (which we call "transpose," "tensor product," and "detensor"—denoted by ^{t}, $\odot$, and $\natural$, respectively) that one does not have with words. However, one does have such operations for the so-called "predicate-abstraction problems" (an important family of dataflow-analysis problems used in essentially all modern-day software model checkers). In predicate-abstraction problems,

- a semiring value is a square Boolean matrix
- the extend operation is Boolean matrix multiplication
- the combine operation is pointwise "or"
- transpose is matrix transpose, and

- tensor product is Kronecker product of square Boolean matrices

The key step is to take each equation of the form "$Y = a \otimes Y \otimes b$" and turn it into "$Z = Z \otimes_{\mathcal{T}} (a^t \odot b)$," where $\otimes_{\mathcal{T}}$ denotes the extend operation in the domain of tensored values. (For predicate abstraction, $\otimes_{\mathcal{T}}$ is Boolean matrix multiplication of tensored matrices.) When this transformation is performed on all equation right-hand sides, the resulting equation system over Z is left-linear, and hence describes a set of paths in a regular language. Consequently, it can be solved by means of Tarjan's path-expression method.

What is not immediately obvious is that from the least-fixed point of the Z system one can obtain the least-fixed point of the Y system—i.e., the Z system can be used to solve the Y system with no loss of precision. The intuition behind the transformation is that linear paths in the Z system mimic derivation trees in the linear context-free grammar of the Y system; as we follow a path in the Z system along edges labeled with, e.g., first $(a_1^t \odot b_1)$ and then $(a_2^t \odot b_2)$, we obtain

$$\begin{aligned}(a_1^t \odot b_1) \otimes_{\mathcal{T}} (a_2^t \odot b_2) &= (a_1^t \otimes a_2^t) \odot (b_2 \otimes b_1) \\ &= \underbrace{(a_1 \otimes \underbrace{a_2}) ^t \odot (\underbrace{b_2} \otimes b_1)}\end{aligned}$$

which produces the kind of mirrored symmetry that we need to track properly the values that arise in the Y system, which have such symmetric correlations (cf. Eqn. (12)). More precisely, the detensor operation performs

$$\lightning (a^t \odot b) = a \otimes b,$$

so that when $\lightning$ is applied to the path's value, we obtain

$$\lightning ((a_1 \otimes a_2)^t \odot (b_2 \otimes b_1)) = \underbrace{a_1 \otimes \underbrace{a_2 \otimes b_2} \otimes b_1},$$

which has the mirrored symmetry found in the values of derivation trees in the Y system. The Z system's paths encode all and only the derivation trees of the Y system.

To use this idea to solve an LCFL equation system precisely, there is one further requirement: the $\lightning$ operation must distribute over the tensored-sum operation. This property causes the detensor of the sum-over-all-Z-paths to equal the desired sum-over-all-Y-tree-valuations. It turns out that such a distributive detensor operation exists for Kronecker products of Boolean matrices, and thus all the pieces fit together for the predicate-abstraction problems. Full details can be found in [12].

Acknowledgments

This work was supported, in part, by a gift from Rajiv and Ritu Batra; DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

References

1. T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Spin Workshop*, 2000.
2. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
3. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.
4. J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian program analysis. *J. ACM*, 57(6), 2010.
5. K. Etessami and M. Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1), 2009.
6. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
7. M. Hopkins and D. Kozen. Parikh's theorem in commutative Kleene algebra. In *LICS*, 1999.
8. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
9. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
10. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
11. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
12. T. Reps, E. Turetsky, and P. Prabhu. Newtonian program analysis via tensor product. *TOPLAS.*, 39(2), 2017.
13. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
14. R. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
15. R. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.