Interprocedural Dataflow Analysis, Part III: Newtonian Program Analysis Via Tensor Product CS701

Thomas Reps

Abstract

In this lecture, we summarize the interprocedural-dataflow-analysis method of Esparza et al., which generalizes Newton's method from a numerical-analysis algorithm for finding roots of real-valued functions to a method for finding fixed-points of systems of equations over semirings. As in its real-valued counterpart, each iteration of the method of Esparza et al. solves a simpler "linearized" problem.

We then move on to a recent method of Reps et al., which attempts to provide an improved technique for solving the linearized problems produced during successive rounds of Newton's method for semirings.

1 Newtonian Program Analysis (NPA)

In previous lectures, we described how interprocedural dataflow-analysis problems could be formalized as path problems in which there are two operators, extend (denoted by $\otimes$) and combine (denoted by $\oplus$). Technically, we are dealing with a *semiring*, which can be defined as follows:

Definition 1.1 *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**.)*

If fact, we need the semiring to have some additional properties—chiefly so that it makes sense to define a **Kleene-star operator** in the following way: $^*\colon D \to D$ is the operation $a^* = \bigoplus_{i \in \mathbb{N}} a^i$, where a^i denotes the i^{th} term in the sequence $a^0 = \underline{1}$ and $a^{i+1} = a^i \otimes a$. See App. A for the technical details.

Our 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 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 $a, b \in D$, $a \otimes b = b \otimes a$. We work with ***non-commutative*** semirings, and henceforth use the term "semiring"—and symbol $\mathcal{S}$—to mean an ***idempotent, non-commutative, (ω-continuous) semiring***.

To simplify notation, we sometimes abbreviate $a \otimes b$ as ab, and we assume the following precedences for operators: $^* > \otimes > \oplus$. We also sometimes use $a \in \mathcal{S}$ rather than $a \in D$.

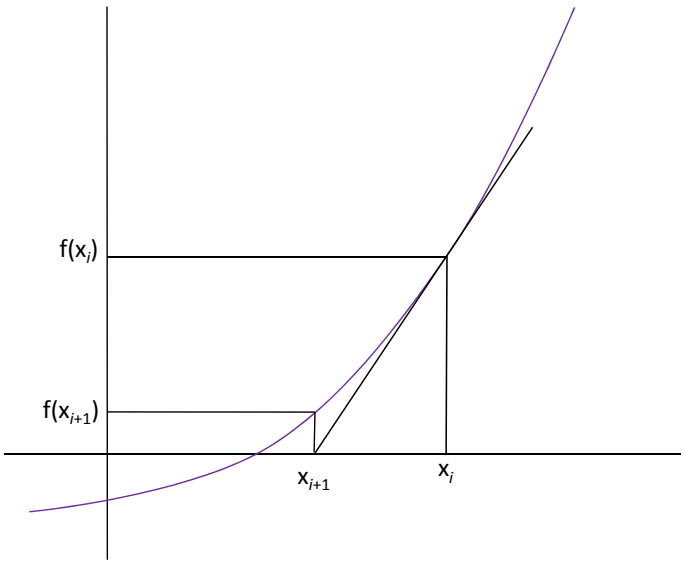

Figure 1: The principle behind Newton's method.

Newton's Method for Root Finding. The jumping-off point for Esparza et al. is the observation that the algorithms for solving program-analysis problems are based on ***Kleene iteration***—i.e., the technique for finding the least fixed-point of $\vec{X} = \vec{f}(\vec{X})$ via the sequence $\vec{\kappa}^{(0)} = \underline{0}$; $\vec{\kappa}^{(i+i)} = \vec{f}(\vec{\kappa}^{(i)})$—whereas in numerical problems, the workhorse for successive-approximation algorithms is Newton's method. The advantage of Newton's method in numerical problems is that when it converges, it converges much faster than Kleene iteration.

Fig. 1 shows how Newton's method can (sometimes) help identify where a root of an expression 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.

Newton's Method for Program Analysis. Esparza et al. [2, 3] have given a generalization of Newton's method that finds the least fixed-point of a system of equations over a semiring. In this section, we summarize their NPA method for the case of idempotent, non-commutative, ω-continuous semirings. Compared to the numerical setting, they have two points that they need to finesse:

1. With numerical functions, the linear model is defined using derivatives and limits. We have no such entities in semirings.
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.

The method of Esparza et al., which they call ***Newtonian Program Analysis (NPA)***, is also an iterative successive-approximation method, but uses the following scheme:[1]

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

[1]For reasons that are immaterial to this discussion, Esparza et al. start the iteration via $\vec{\nu}^{(0)} = \langle f_1(\bot), \ldots, f_n(\bot) \rangle$ rather than $\vec{\nu}^{(0)} = \vec{\bot}$. Our goal here is to bring out the essential similarities between Kleene iteration and NPA.

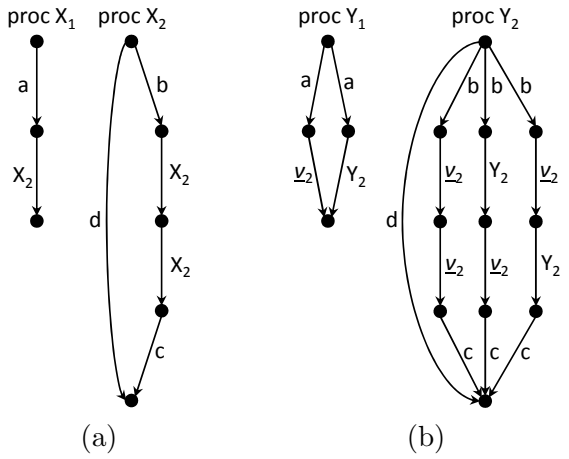

Figure 2: (a) Graphical depiction of the equation system given in Eqn. (2) as an interprocedural control-flow graph. The three edges labeled "X_2" represent calls to procedure X_2. (b) Linearized equation system over $\vec{Y}$ obtained from Eqn. (2) via Eqn. (4).

where LinearCorrectionTerm$(\vec{f}, \vec{\nu}^{(i)})$ is a correction term—a function of $\vec{f}$ and the current approximation $\vec{\nu}^{(i)}$—that nudges the next approximation $\vec{\nu}^{(i+1)}$ in the right direction at each step. The sense in which the correction term is "linear" will be discussed shortly, but it is that linearity property that makes it proper to say that Eqn. (1) is a form of Newton's method.

Example 1.2 *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 semiring that captures a suitable abstraction of the program's actions (such as the relational weight domain). Let a, b, c, and d denote the semiring elements that abstract 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:

$$X_1 = a \otimes X_2 \qquad X_2 = d \oplus b \otimes X_2 \otimes X_2 \otimes c. \tag{2}$$

An equation system can also be viewed as a representation of a program's interprocedural control-flow graph (CFG). See Fig. 2(a).

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 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 system of polynomial equations has the form

$$X_1 = f_1(X_1, \ldots, X_n) \quad \cdots \quad X_n = f_n(X_1, \ldots, X_n),$$

or equivalently, $\vec{X} = \vec{f}(\vec{X})$, where $\vec{X} = \langle X_1, \ldots, X_n \rangle$ and $\vec{f} = \lambda \vec{X}.\langle f_1(\vec{X}), \ldots, f_n(\vec{X}) \rangle$. For instance, for Eqn. (2), $\vec{f} \stackrel{\text{def}}{=} \lambda \vec{X}.\langle a \otimes X_2, d \oplus b \otimes X_2 \otimes X_2 \otimes c \rangle$.

Kleene iteration is the well-known technique for finding the least fixed-point of $\vec{X} = \vec{f}(\vec{X})$ via the sequence $\vec{\kappa}^{(0)} = \underline{0}$; $\vec{\kappa}^{(i+i)} = \vec{f}(\vec{\kappa}^{(i)})$. Esparza et al. [2, 3] devised an alternative method, called NPA, for finding the least fixed-point of $\vec{X} = \vec{f}(\vec{X})$. With NPA, one solves the following sequence of problems for $\vec{\nu}$:

$$\boxed{\begin{array}{rcl} \vec{\nu}^{(0)} & = & (f_1(\vec{\underline{0}}), \ldots, f_n(\vec{\underline{0}})) \\ \vec{\nu}^{(i+1)} & = & \vec{Y}^{(i)} \end{array}} \tag{3}$$

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

$$\boxed{\vec{Y} = \vec{f}(\vec{\nu}^{(i)}) \oplus \mathcal{D}\vec{f}|_{\vec{\nu}^{(i)}}(\vec{Y})} \tag{4}$$

and $\mathcal{D}\vec{f}|_{\vec{\nu}^{(i)}}(\vec{Y})$ is the ***multivariate differential*** of $\vec{f}$ at $\vec{\nu}^{(i)}$, defined below (see Defn. 1.3). Eqns. (3) and (4) resemble Kleene iteration, except that on each iteration $\vec{f}(\vec{\nu}^{(i)})$ is "corrected" by the amount $\mathcal{D}\vec{f}|_{\vec{\nu}^{(i)}}(\vec{Y})$.

There is a close analogy between NPA and the use of Newton's method in numerical analysis to solve a system of polynomial equations $\vec{f}(\vec{X}) = \vec{0}$. In both cases, one creates a linear approximation of $\vec{f}$ around the point $(\vec{\nu}^{(i)}, \vec{f}(\vec{\nu}^{(i)}))$, and then uses the solution of the linear system in the next approximation of $\vec{X}$. The sequence $\vec{\nu}^{(0)}, \vec{\nu}^{(1)}, \ldots, \vec{\nu}^{(i)}, \ldots$ is called the ***Newton sequence*** for $\vec{X} = \vec{f}(\vec{X})$. The process of solving Eqns. (3) and (4) for $\vec{\nu}^{(i+1)}$, given $\vec{\nu}^{(i)}$, is called a ***Newton step*** or one ***Newton round***. For polynomial equations over a semiring, the linear approximation of $\vec{f}$ is created as follows:

Definition 1.3 *[2, 3] Let $f_i(\vec{X})$ be a component function of $\vec{f}(\vec{X})$. The **differential** of $f_i(\vec{X})$ with respect to X_j at $\underline{\vec{\nu}}$, denoted by $\mathcal{D}_{X_j} f_i|_{\underline{\vec{\nu}}}(\vec{Y})$, is defined as follows:*

$$\boxed{\mathcal{D}_{X_j} f_i|_{\underline{\vec{\nu}}}(\vec{Y}) = \begin{cases} \underline{0} & \text{if } f_i = s \in \mathcal{S} \\ \underline{0} & \text{if } f_i = X_k \text{ and } k \neq j \\ Y_j & \text{if } f_i = X_j \\ \bigoplus\limits_{k \in K} \mathcal{D}_{X_j} g_k|_{\underline{\vec{\nu}}}(\vec{Y}) & \text{if } f_i = \bigoplus\limits_{k \in K} g_k \\ \left(\begin{array}{c} \mathcal{D}_{X_j} g|_{\underline{\vec{\nu}}}(\vec{Y}) \otimes h(\underline{\vec{\nu}}) \\ \oplus\, g(\underline{\vec{\nu}}) \otimes \mathcal{D}_{X_j} h|_{\underline{\vec{\nu}}}(\vec{Y}) \end{array} \right) & \text{if } f_i = g \otimes h \end{cases}} \tag{5}$$

*Let $\vec{f}$ be a multivariate polynomial function defined by $\vec{f} \stackrel{\text{def}}{=} \lambda \vec{X}.(f_1(\vec{X}), \ldots, f_n(\vec{X}))$. The **multivariate differential** of $\vec{f}$ at $\underline{\vec{\nu}}$, denoted by $\mathcal{D}\vec{f}|_{\underline{\vec{\nu}}}(\vec{Y})$, is defined as follows:*

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

$\mathcal{D} f_i|_{\underline{\vec{\nu}}}(\vec{Y})$ denotes the i^{th} component of $\mathcal{D}\vec{f}|_{\underline{\vec{\nu}}}(\vec{Y})$.

The fourth case in Eqn. (5) generalizes the differential of a binary combine, i.e.,

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

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

$$\frac{d}{dx}(g * h) = \frac{dg}{dx} * h + g * \frac{dg}{dx}, \tag{6}$$

and in particular the differential form of the product rule:

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

We refer to the creation of Eqn. (4) from $\vec{X} = \vec{f}(\vec{X})$ as the ***NPA linearizing transformation***.

Example 1.4 *For Eqn. (2), the multivariate differential of $\vec{f}$ at the value $\underline{\vec{\nu}} = \langle \underline{\nu}_1, \underline{\nu}_2 \rangle$ is*

$$\begin{aligned}
\mathcal{D}\vec{f}|_{(\underline{\nu}_1, \underline{\nu}_2)}(\vec{Y}) &= \left\langle \begin{matrix} \mathcal{D}_{X_1} f_1|_{(\underline{\nu}_1, \underline{\nu}_2)}(\vec{Y}) \oplus \mathcal{D}_{X_2} f_1|_{(\underline{\nu}_1, \underline{\nu}_2)}(\vec{Y}), \\ \mathcal{D}_{X_1} f_2|_{(\underline{\nu}_1, \underline{\nu}_2)}(\vec{Y}) \oplus \mathcal{D}_{X_2} f_2|_{(\underline{\nu}_1, \underline{\nu}_2)}(\vec{Y}) \end{matrix} \right\rangle \\
&= \left\langle \underline{0} \oplus a \otimes Y_2, \underline{0} \oplus \begin{pmatrix} b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \right\rangle \\
&= \left\langle a \otimes Y_2, \begin{pmatrix} b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \right\rangle
\end{aligned} \tag{7}$$

From Eqn. (4), we then obtain the following linearized system of equations, which is also depicted graphically in Fig. 2(b):

$$\langle Y_1, Y_2 \rangle = \left\langle \begin{pmatrix} a \otimes \underline{\nu}_2 \\ \oplus\ a \otimes Y_2 \end{pmatrix}, \begin{pmatrix} d \\ \oplus\ b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \\ \oplus\ b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c \end{pmatrix} \right\rangle \tag{8}$$

On the $i+1^{st}$ Newton round, we need to solve Eqn. (8) 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 $\langle \nu_1^{(i+1)}, \nu_2^{(i+1)} \rangle \leftarrow \langle Y_1, Y_2 \rangle$.

It is instructive to consider why the second component in Eqn. (7) has the form that it has:

$$b \otimes Y_2 \otimes \underline{\nu}_2 \otimes c \oplus b \otimes \underline{\nu}_2 \otimes Y_2 \otimes c.$$

Consider differentiating the numeric expression "$fXgXh$" with respect to X. You'll probably immediately say $fXgXh = fghX^2$, and therefore the derivative is $2fghX$. But let's go more by what Eqn. (6) says:

$$\begin{aligned}
\frac{d}{dX}(fXgXh) &= f * \frac{dX}{dX} * g * X * h + f * X * g * \frac{dX}{dX} * h \\
&= f * 1 * g * X * h + f * X * g * 1 * h \\
&= fgXh + fXgh
\end{aligned}$$

If multiplication is not commutative (which is the case with a semiring's $\otimes$ operation), then you cannot simplify the last line to $2fghX$.

Note that Eqn. (8) is a linear problem in that each summand has just a single occurrence of a Y_i. Fig. 2(b) gives a graphical depiction of the linearized problem.

A Misconception on My Part. While Prathmesh Prabhu, a student in CS704, was presenting the experiment that he and two other students had done with NPA as a class project, I formulated the following analogy in my mind:

$$\begin{array}{rcl} \text{Polynomial} & \rightarrow & \text{Linear} \\ \text{Interprocedural} & \rightarrow & \text{Intraprocedural} \end{array}$$

In other words, I was thinking that solving each linearized problem corresponds to solving an ***intra***procedural dataflow-analysis problem—a topic that has a fifty-year history [12, 7, 6, 5, 11]. In particular, Tarjan's path-expression method [10] finds a regular expression for each of the variables in a set of mutually recursive left-linear equations. The regular expressions are then evaluated using an appropriate interpretation of the regular operators $+$, $\cdot$, and *. Moreover, Prathmesh's study indicated that (i) NPA was not an improvement over conventional methods for interprocedural dataflow analysis, and (ii) 98% of the time was spent performing classical fixed-point iteration to solve the linearized problem. Why not just apply a fast intraprocedural solver?

Prathmesh's reply set me back; he pointed out what we mentioned above, namely, when Newton's method is used in numerical-analysis problems, ***commutativity of multiplication*** is relied on to rearrange an expression of the form "$c * X + X * d$" in the linearized problem into one of the form "$c*X+d*X$," which equals "$(c+d)*X$." In contrast, in interprocedural dataflow analysis, a dataflow value is typically an abstract transformer (i.e., it represents a function from sets of states to sets of states) [1, 9]. Consequently, the "multiplication" operation is typically the reversal of function composition—$v_1 * v_2 \stackrel{\text{def}}{=} v_2 \circ v_1$—which is not a commutative operation. When NPA is used with a non-commutative semiring, an expression "$c * X + X * d$" in the linearized problem cannot be rearranged: coefficients can appear on both sides of variables.

From a formal-languages perspective, the linearized equation systems that arise in numerical analysis correspond to path problems described by ***regular languages***. However, when expressions of the form "$c*X+X*d$" cannot be rearranged, the linearized equation systems correspond to path problems described by ***linear context-free languages (LCFLs)***. Conventional intraprocedural dataflow-analysis algorithms solve only regular-language path problems, and hence cannot, in general, be applied to the linearized equation systems considered on each round of NPA. Consequently, we are stuck performing classical fixed-point iteration on the LCFL equation systems. (Applying NPA's linearization transformation to one of the LCFL equation systems just results in the same LCFL equation system, and so one would not make any progress.)

On the face of it, it seems impossible, therefore, that Tarjan's method could help in any way: formal-language theory tells us that LCFL $\supsetneq$ Regular. In particular, the canonical example of a non-regular language, $\{b^i c^i \mid i \in \mathbb{N}\}$, is an LCFL.

However, the remainder of the lecture describes how—despite this obstacle—there are non-commutative semirings for which we can transform the problem so that Tarjan's method applies. Moreover, one of the families of semirings for which our transformation applies is the set of ***predicate-abstraction domains*** [4], which are the foundation of most of today's software model checkers.

A Promising Step: Pairing. A left-linear equation system corresponds to a left-linear grammar, and hence a regular language. The fact that Tarjan's path-expression method [10] 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 L that arises on each Newton round—i.e., transform L into a left-linear equation system L_{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 is 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, if we read the Y_2 component of Eqn. (8) as a grammar,

$$\begin{array}{rcl} Y_2 & ::= & d \\ & | & b\ \underline{\nu}_2\ \underline{\nu}_2\ c \\ & | & b\ Y_2\ \underline{\nu}_2\ c \\ & | & b\ \underline{\nu}_2\ Y_2\ c, \end{array}$$

it resembles the language $\mathcal{L} = \{b^i c^i \mid i \in \mathbb{N}\}$, which is the canonical example of an LCFL that is not regular. $\mathcal{L}$ can be defined via the linear context-free grammar

$$S ::= \epsilon \mid b\ S\ c \tag{9}$$

in which the second production allows matching b's and c's to be accumulated on the left and right sides of nonterminal S. Moreover, if grammar (9) is extended to have K matching rules

$$S ::= \epsilon \mid b_j\ S\ c_j \quad 1 \leq j \leq K \tag{10}$$

the generated strings have bilateral symmetry, e.g.,

$$\ldots \underbrace{b_2\, \underbrace{b_1 c_1}\, c_2} \ldots$$

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. This observation suggests the strategy of using ***pairs*** in which left-side and right-side values are accumulated separately but concurrently, so that the desired correlation is maintained. Toward this end, we define extend and combine on pairs as follows:

$$(a_1, b_1) \otimes_p (a_2, b_2) = (a_2 \otimes a_1, b_1 \otimes b_2) \tag{11}$$

$$(a_1, b_1) \oplus_p (a_2, b_2) = (a_1 \oplus a_2, b_1 \oplus b_2) \tag{12}$$

Note the order-reversal in the first component of the right-hand side of Eqn. (11): "$a_2 \otimes a_1$."

Given a pair (a, b), we can read out a normal value via the operation $\mathcal{R}(a, b) \stackrel{\text{def}}{=} a \otimes b$. Because of the order-reversal in Eqn. (11), we have

$$\begin{array}{rcl} \mathcal{R}((a_1, b_1) \otimes_p (a_2, b_2)) & = & \mathcal{R}((a_2 \otimes a_1, b_1 \otimes b_2)) \\ & = & \underbrace{a_2 \otimes \underbrace{a_1 \otimes b_1} \otimes b_2}\,. \end{array}$$

The braces highlight the fact that we have achieved the desired mirrored matching of (i) a_1 with b_1, and (ii) a_2 with b_2.

Example 1.5 *Using $\otimes_p$ and $\oplus_p$, we can transform a linear equation (and more generally a set of linear equations) by pairing semiring values that appear to the left of a variable with the values that appear to the right of the variable, placing the pair to the variable's right. For instance, the equation for Y_2 in Eqn. (8) is transformed into*

$$Z_2 = \begin{pmatrix} & (\underline{1}, d) \\ \oplus_p & (\underline{1}, b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c) \\ \oplus_p & Z_2 \otimes_p (b, \underline{\nu}_2 \otimes c) \\ \oplus_p & Z_2 \otimes_p (b \otimes \underline{\nu}_2, c) \end{pmatrix} \tag{13}$$

*where Z_2 is now a variable that takes on **pairs** of semiring values. After collecting terms, we have an equation of the form*

$$Z_2 = A \oplus_p Z_2 \otimes_p B, \tag{14}$$

$$\textit{where} \quad A = (\underline{1}, d \oplus b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c), \tag{15}$$

$$\textit{and} \quad B = (b \oplus b \otimes \underline{\nu}_2, \underline{\nu}_2 \otimes c \oplus c). \tag{16}$$

Eqn. (14) is similar to the equation over formal languages

$$Z_2 = A + (Z_2 \cdot B),$$

for which the regular expression $A \cdot B^*$ is a closed-form solution for Z_2. Similarly, the solution of Eqn. (14) for Z_2 over paired semiring values is given by

$$Z_2 = A \otimes_p B^{*_p}, \tag{17}$$

where B^{*_p} denotes $\bigoplus_{i \in \mathbb{N}}^{\mathrm{p}} B^i$ (in which the repeated "multiplication" operation in B^i is $\otimes_p$). If the answer obtained for Z_2 is the pair (w_1, w_2), we can read out the value for Z_2 as $\mathcal{R}((w_1, w_2)) = w_1 \otimes w_2$.

The algorithm demonstrated above can be stated as follows:

Algorithm 1.6 *To solve a linear equation system L,*

1. *Convert L into a left-linear equation system L_{Reg} (with weights that consist of pairs of semiring values).*
2. *Find the least solution of equation system L_{Reg}.*
3. *Apply the readout operation $\mathcal{R}$ to the least solution of L_{Reg} to obtain a solution to L.*

In our example, for step (2) we expressed the least solution of Eqn. (14) in closed form, as a regular expression (Eqn. (36)), which means that the solution for Z_2 can be obtained merely by evaluating the regular expression. In general, when equation system L_{Reg} has a larger number of variables, for step (2) we can use Tarjan's path-expression method [10], which finds a regular expression for each of the variables in a set of mutually recursive left-linear equations.

This approach has a lot of promise for Newtonian program analysis because the structure of L_{Reg}—***and hence of the corresponding regular expressions***—remains fixed from round to round. Consequently, we only need to perform the expensive step of regular-expression construction via Tarjan's method once, before the first round. The actions taken for step (2) on each Newton round are as follows: (i) in each regular expression, replace the constant-valued leaves $\{\underline{\nu}_i\}$, which represent previous-round values, with updated constants, and (ii) reevaluate the regular expression.

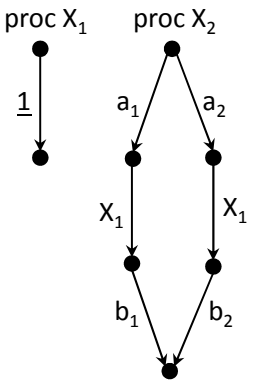

Figure 3: A path problem that has exactly two paths.

In our example, the original linearized system of Eqn. (8), transformed to left-linear form, is

$$\langle Z_1, Z_2 \rangle = \langle (\underline{1}, a \otimes \underline{\nu}_2) \oplus_p Z_2 \otimes_p (a, \underline{1}), A \oplus_p Z_2 \otimes_p B \rangle,$$

for which we have the closed-form solution

$$\langle Z_1, Z_2 \rangle = \left\langle \begin{matrix} (\underline{1}, a \otimes \underline{\nu}_2) \oplus_p A \otimes_p B^{*_p} \otimes_p (a, \underline{1}), \\ A \otimes_p B^{*_p} \end{matrix} \right\rangle. \tag{18}$$

To solve the original system of equations given in Eqn. (2),

1. First, set $\underline{\nu}_2$ to $\underline{0}$ in Eqn. (18) and evaluate the right-hand side:

$$\langle Z_1, Z_2 \rangle = \left\langle \begin{matrix} (\underline{1}, \underline{0}) \oplus_p (\underline{1}, d) \otimes_p (b, c)^{*_p} \otimes_p (a, \underline{1}), \\ (\underline{1}, d) \otimes_p (b, c)^{*_p} \end{matrix} \right\rangle. \tag{19}$$

2. Then, until convergence, repeat the following steps:
   (a) Apply $\mathcal{R}$ to the value obtained for Z_2 to obtain the value of $\underline{\nu}_2$ to use during the next round.
   (b) Use that value in Eqns. (15) and (16), and evaluate the right-hand side of Eqn. (18) to obtain new values for Z_1 and Z_2.

Pairing Fails to Deliver. Unfortunately, the method given as Alg. 1.6 is not guaranteed to produce the desired least-fixed-point solution to an LCFL equation system L. The reason is that the read-out operation $\mathcal{R}$ does not, in general, distribute over $\oplus_p$. Consider the equation system

$$X_1 = \underline{1} \qquad X_2 = a_1 X_1 b_1 \oplus a_2 X_1 b_2,$$

which is depicted graphically in Fig. 3. This system corresponds to a graph with two paths. The least solution for X_2 is $a_1 b_1 \oplus a_2 b_2$, where $a_1 b_1$ and $a_2 b_2$ are the contributions from the two paths. However, when treated as a paired-semiring-value problem, we have

$$Z_1 = (\underline{1}, \underline{1}) \qquad Z_2 = Z_1 \otimes_p ((a_1, b_1) \oplus_p (a_2, b_2)).$$

The least solution for Z_2 is $(a_1, b_1) \oplus_p (a_2, b_2)$, whose readout value is $\mathcal{R}((a_1, b_1) \oplus_p (a_2, b_2))$. However, the latter does not equal $a_1 b_1 \oplus a_2 b_2$.

$$\begin{aligned} \mathcal{R}((a_1, b_1) \oplus_p (a_2, b_2)) &= \mathcal{R}((a_1 \oplus a_2, b_1 \oplus b_2)) \\ &= (a_1 \oplus a_2) \otimes (b_1 \oplus b_2) \\ &= a_1 b_1 \oplus a_2 b_1 \oplus a_1 b_2 \oplus a_2 b_2 \\ &\sqsupseteq a_1 b_1 \oplus a_2 b_2 \\ &= \mathcal{R}((a_1, b_1)) \oplus \mathcal{R}((a_2, b_2)). \end{aligned} \tag{20}$$

In other words, using combines of pairs leads to cross-terms, such as $a_2 b_1$ and $a_1 b_2$, and consequently answers obtained by (i) solving Eqn. (14) over paired semiring values for the combine-over-all-values answer, and (ii) applying $\mathcal{R}$ to the result, could return an overapproximation ($\sqsupset$) of the least solution of the original LCFL equation system L.

A Different Kind of Pairing. In light of the example presented above, the prospects for harnessing Tarjan's path-expression method for use during NPA look rather bleak. However, there is still one glimmer of hope:

> A transformation of the linearized problem to left-linear form is not actually forced to use ***pairing***: given a "coupled value" $c = (a, b)$, we never need to recover from c the value of either a or b alone; we only need to be able to obtain the value $a \otimes b$.

Thus, by using some other binary operator to couple values together, it may still be possible to perform a transformation similar to the conversion of the equation for Y_2 in Eqn. (8) into Eqn. (13). Of course, the final answer read out of the solution to the left-linear problem must not have contributions from undesired cross-terms.

Thus, we must ask the question,

> Is there an operation to couple two values, a and b, that returns a "blended" value in which a and b are hidden, but the product $a \otimes b$ is recoverable.

For instance, in the RSA cryptography, the product ab of two primes a and b is a kind of "blended" value that hides a and b (unless a great deal of computational effort is expended), but one can trivially recover the product ab.

It turns out that the answer to our question is "yes" (!), at least for some semirings of interest. We will explain the principle for the IFDS framework [8] and predicate-abstraction domains [4] before giving a more abstract definition of the required properties.

The semiring values for both the IFDS framework and predicate abstraction can be represented as Boolean matrices. $N \times N$ Boolean matrices support an operation called the ***Kronecker product***, denoted by $\odot$, defined as follows:

$$R \odot S = \begin{bmatrix} r_{1,1}S & \cdots & r_{1,N}S \\ \vdots & \ddots & \vdots \\ r_{N,1}S & \cdots & r_{N,N}S \end{bmatrix}$$

which yields an $N^2 \times N^2$ binary matrix whose entries are

$$(R \odot S)[(a-1)N + b, (a'-1)N + b'] = R(a, a') \wedge S(b, b').$$

Let's consider why the Kronecker product of two Boolean matrices A and B allows their product $A \times B$ to be recovered later:

$$\begin{aligned} A \odot B &= \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \odot \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \\ &= \left[\begin{array}{cc|cc} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,2}b_{1,1} & a_{1,2}b_{2,2} \\ \hline a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} \end{array}\right] \end{aligned} \tag{21}$$

$$\begin{aligned} A \times B &= \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} a_{1,1}b_{1,1} \vee a_{1,2}b_{2,1} & a_{1,1}b_{1,2} \vee a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} \vee a_{2,2}b_{2,1} & a_{2,1}b_{1,2} \vee a_{2,2}b_{2,2} \end{bmatrix} \end{aligned} \tag{22}$$

You can see the each of the summands in Eqn. (22) is found in Eqn. (21). It is not too difficult to create an expression that computes $A \times B$ from $A \odot B$. (Hint: It involves multiplying $A \odot B$ on both the left-hand and right-hand sides by suitable non-square matrices.) Moreover, it turns out that this "readout" operations distributes over the $\oplus$ operation for Boolean matrices, which is component-wise "or." This latter property is crucial to avoiding the kinds of cross-terms that cropped up when we tried to use pairing.

We define the desired "coupling" operation in terms of two primitives: ***transpose*** and ***tensor product***:

Definition 1.7 *Let $\mathcal{S} = (D, \oplus, \otimes, \underline{0}, \underline{1})$ be a semiring. $\mathcal{S}$ has a **transpose** operation, denoted by $\cdot^t : D \to D$, if for all elements $a, a_1, a_2 \in D$ the following properties hold:*

$$(a_1 \otimes a_2)^t = a_2^t \otimes a_1^t \tag{23}$$
$$(a_1 \oplus a_2)^t = a_1^t \oplus a_2^t \tag{24}$$
$$(a^t)^t = a. \tag{25}$$

*A **tensor-product** semiring over $\mathcal{S}$ is defined to be another semiring $\mathcal{S}_\mathcal{T} = (D_\mathcal{T}, \oplus_\mathcal{T}, \otimes_\mathcal{T}, \underline{0}_\mathcal{T}, \underline{1}_\mathcal{T})$, where $\mathcal{S}$ and $\mathcal{S}_\mathcal{T}$ support a **tensor-product** operation, denoted by $\odot : D \times D \to D_\mathcal{T}$, such that for all $a, a_1, a_2, b_1, b_2, c_1, c_2 \in D$, the following properties hold:*

$$\underline{0} \odot a = a \odot \underline{0} \quad = \quad \underline{0}_\mathcal{T} \tag{26}$$
$$a_1 \odot (b_2 \oplus c_2) = (a_1 \odot b_2) \oplus_\mathcal{T} (a_1 \odot c_2) \tag{27}$$
$$(b_1 \oplus c_1) \odot a_2 = (b_1 \odot a_2) \oplus_\mathcal{T} (c_1 \odot a_2) \tag{28}$$
$$(a_1 \odot b_1) \otimes_\mathcal{T} (a_2 \odot b_2) = (a_1 \otimes a_2) \odot (b_1 \otimes b_2). \tag{29}$$

*A tensor-product semiring defined over a semiring with transpose has a **(sequential) detensor-transpose** operation, denoted by $\lightning^{(t,\cdot)} : D_\mathcal{T} \to D$, if for all elements $a_1, a_2 \in D$ and $p_1, p_2 \in D_\mathcal{T}$ the following properties hold:*

$$\lightning^{(t,\cdot)}(a_1 \odot a_2) = (a_1^t \otimes a_2) \tag{30}$$
$$\lightning^{(t,\cdot)}(p_1 \oplus_\mathcal{T} p_2) = \lightning^{(t,\cdot)}(p_1) \oplus \lightning^{(t,\cdot)}(p_2). \tag{31}$$

We assume that Eqns. (24), (27), (28), and (31) also hold for infinite combines.

*For brevity, we say that $\mathcal{S}$ is an **admissible semiring** if (i) $\mathcal{S}$ has a transpose operation, (ii) $\mathcal{S}$ has an associated tensor-product semiring $\mathcal{S}_\mathcal{T}$, and (iii) $\mathcal{S}_\mathcal{T}$ has a sequential detensor-transpose operation. Henceforth, we consider only admissible semirings.*

*The operation to **couple** pairs of values from an admissible semiring, denoted by $\mathcal{C} : D \times D \to D_\mathcal{T}$, is defined as follows:*

$$\mathcal{C}(a, b) \stackrel{\text{def}}{=} (a^t \odot b).$$

Note that by Eqns. (23) and (29),

$$\begin{aligned} \mathcal{C}(a_1, b_1) \otimes_\mathcal{T} \mathcal{C}(a_2, b_2) &= (a_1^t \odot b_1) \otimes_\mathcal{T} (a_2^t \odot b_2) \\ &= (a_1^t \otimes a_2^t) \odot (b_1 \otimes b_2) \\ &= (a_2 \otimes a_1)^t \odot (b_1 \otimes b_2) \\ &= \mathcal{C}(a_2 \otimes a_1, b_1 \otimes b_2) \end{aligned} \tag{32}$$

The order-reversal vis à vis $\otimes_\mathcal{T}$ and $\otimes$ in Eqn. (32) will substitute for the order-reversal vis à vis $\otimes_p$ and $\otimes$ in Eqn. (11).

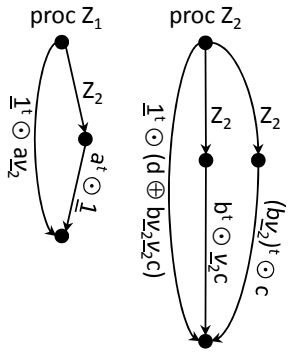

Figure 4: Graphical representation of the linearized equation system over $\vec{Z}$ obtained from Eqn. (2) using tensor-product-based coupling.

The operator that plays the role of $\mathcal{R}$ is $\lightning^{(t,\cdot)}$. The superscript in $\lightning^{(t,\cdot)}$ serves as a reminder that Eqn. (30) performs an additional transpose on the first argument of a coupled value $(a^t \odot b)$, so that $\lightning^{(t,\cdot)}(a^t \odot b)$ becomes $(a^t)^t \otimes b = a \otimes b$. Consequently,

$$\begin{aligned}\lightning^{(t,\cdot)}(\mathcal{C}(a_2 \otimes a_1, b_1 \otimes b_2)) &= \lightning^{(t,\cdot)}((a_2 \otimes a_1)^t \odot (b_1 \otimes b_2)) \\ &= ((a_2 \otimes a_1)^t)^t \otimes (b_1 \otimes b_2) \\ &= \underbrace{a_2 \otimes \underbrace{a_1 \otimes b_1}\otimes b_2}\end{aligned}$$

which has the desired matching of a_1 with b_1 and a_2 with b_2. Moreover, in contrast with Eqn. (20), by Eqn. (31), $\lightning^{(t,\cdot)}$ does not produce cross-terms:

$$\begin{aligned}\lightning^{(t,\cdot)}((a_1^t \odot b_1) \oplus_{\mathcal{T}} (a_2^t \odot b_2)) &= \lightning^{(t,\cdot)}(a_1^t \odot b_1) \oplus_{\mathcal{T}} \lightning^{(t,\cdot)}(a_2^t \odot b_2) \\ &= a_1 b_1 \oplus a_2 b_2.\end{aligned}$$

Example 1.8 *Using tensor-product-based coupling, the Y_2 component of Eqn. (8) would be transformed into*

$$Z_2 = \begin{pmatrix} (\underline{1}^t \odot (d \oplus b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c)) \\ \oplus_{\mathcal{T}}\ Z_2 \otimes_{\mathcal{T}} (b^t \odot (\underline{\nu}_2 \otimes c)) \\ \oplus_{\mathcal{T}}\ Z_2 \otimes_{\mathcal{T}} ((b \otimes \underline{\nu}_2)^t \odot c) \end{pmatrix} \tag{33}$$

which is depicted in Fig. 4. After collecting terms, we have

$$Z_2 = A \oplus_{\mathcal{T}} (Z_2 \otimes_{\mathcal{T}} B), \tag{34}$$

$$\textit{where}\quad A = (\underline{1}^t \odot (d \oplus b \otimes \underline{\nu}_2 \otimes \underline{\nu}_2 \otimes c))$$

$$\textit{and}\quad B = (b^t \odot (\underline{\nu}_2 \otimes c)) \oplus_{\mathcal{T}} ((b \otimes \underline{\nu}_2)^t \odot c) \tag{35}$$

which has the solution

$$Z_2 = A \otimes_{\mathcal{T}} B^{*_{\mathcal{T}}}. \tag{36}$$

The (untensored) value for X_2 is then obtained as $X_2 = \lightning^{(t,\cdot)}(Z_2)$.

References

[1] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*. North-Holland, 1978.

[2] J. Esparza, S. Kiefer, and M. Luttenberger. Newton's method for omega-continuous semirings. In *Int. Colloq. on Automata, Langs., and Programming*, 2008.
[3] J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian program analysis. *J. ACM*, 57(6), 2010.
[4] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verif.*, 1997.
[5] S.L. Graham and M. Wegman. A fast and usually linear algorithm for data flow analysis. *J. ACM*, 23(1):172–202, 1976.
[6] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.
[7] G.A. Kildall. A unified approach to global program optimization. In *Princ. of Prog. Lang.*, 1973.
[8] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Princ. of Prog. Lang.*, 1995.
[9] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
[10] R.E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
[11] R.E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.
[12] V. Vyssotsky and P. Wegner. A graph theoretical Fortran source language analyzer. Unpublished technical report, Bell Labs, Murray-Hill NJ (as cited in Aho et al., "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1986), 1963.

A Definitions

Definition A.1 *An* **ω-continuous semiring** *is a semiring with the following additional properties:*

1. *The relation* $\sqsubseteq \stackrel{\text{def}}{=} \{(a,b) \in D \times D \mid \exists d\colon a \oplus d = b\}$ *is a partial order.*
2. *Every ω-chain $(a_i)_{i\in\mathbb{N}}$ (i.e., for all $i \in \mathbb{N}$ $a_i \sqsubseteq a_{i+1}$) has a supremum with respect to $\sqsubseteq$, denoted by $\sup_{i\in\mathbb{N}} a_i$.*
3. *Given an arbitrary sequence $(c_i)_{i\in\mathbb{N}}$, define*

$$\bigoplus_{i\in\mathbb{N}} c_i \stackrel{\text{def}}{=} \sup\{c_0 \oplus c_1 \oplus \ldots \oplus c_i \mid i \in \mathbb{N}\}.$$

The supremum exists by (2) above. Then, for every sequence $(a_i)_{i\in\mathbb{N}}$, for every $b \in \mathcal{S}$, and every partition $(I_j)_{j\in J}$ of $\mathbb{N}$, the following properties all hold:

$$b \otimes \left(\bigoplus_{i\in\mathbb{N}} a_i\right) = \bigoplus_{i\in\mathbb{N}} (b \otimes a_i)$$
$$\left(\bigoplus_{i\in\mathbb{N}} a_i\right) \otimes b = \bigoplus_{i\in\mathbb{N}} (a_i \otimes b)$$
$$\bigoplus_{j\in J} \left(\bigoplus_{i\in I_j} a_i\right) = \bigoplus_{i\in\mathbb{N}} a_i$$

The notation a^i denotes the i^{th} term in the sequence in which $a^0 = \underline{1}$ and $a^{i+1} = a^i \otimes a$. An ω-continuous semiring has a ***Kleene-star operator*** $^*\colon D \to D$ *defined as follows:* $a^* = \bigoplus_{i\in\mathbb{N}} a^i$.