FPsolve: A Generic Solver for Fixpoint Equations over Semirings [⋆]

Javier Esparza, Michael Luttenberger, and Maximilian Schlund

Technische Universität München, {esparza,luttenbe,schlund}@in.tum.de

Abstract. We introduce FPsolve, an implementation of generic algorithms for solving fixpoint equations over semirings. We first illustrate the interest of generic solvers by means of a scenario. We then succinctly describe some of the algorithms implemented in the tool, and provide some implementation details.

1 Introduction

We present FPsolve[1], a solver for *algebraic systems of equations* first introduced in [?]. These are systems of equations of the form

$$X_1 = f_1(X_1, X_2, \ldots, X_n) \quad \cdots \quad X_n = f_n(X_1, X_2, \ldots, X_n)$$

where $f_1, \ldots, f_n$ are polynomials in the variables $X_1, X_2, \ldots, X_n$. The coefficients of the polynomials can be elements of any semiring satisfying some weak conditions, which ensure that there exists a unique smallest solution. FPsolve implements a number of *generic* algorithms, i.e. algorithms parametric in the semiring operations of addition and multiplication, plus possibly the Kleene star operation.

Algebraic systems naturally arise in various settings:

– The language of a context-free grammar like $X \to aXX \mid b$ is the least solution of the equation $X = aXX + b$ over the semiring whose elements are languages, with union and concatenation of languages as sum and product.
– Shortest-paths problems on finite graphs and on some infinite graphs, like those generated by weighted pushdown automata, can be reduced to solving fixed-point equations over a semiring having the possible edge weights as elements [7,16].
– Data-flow equations associated to many intra- and interprocedural dataflow analyses are fixed-point equations over complete lattices [13], which can often be recast as equations on semirings [16,6].
– Authorization problems (like, for instance, the authorization problem for the SPKI/SDSI authorization system), can be recast as a reachability problem in weighted pushdown automata [12], and thus to algebraic systems [6].

⋆ This work was funded by the DFG project “Polynomial Systems on Semirings: Foundations, Algorithms, Applications”

1 Freely available from `https://github.com/mschlund/FPsolve`.

– Computing the reputation of a principal in a reputation system (a system in which principals can recommend other principals, and rules are used to compute reputation out of a set of direct recommendations) reduces to solving an algebraic system [5].
– Evaluating a Datalog query can be reformulated as the problem of deciding whether a non-terminal of a context-free grammar is productive or not, and so it also amounts to solving a system of equations. Moreover, several problems concerning the computation of *provenance information*, an important research topic in database theory, reduces to solving an associated algebraic system over different semirings. [11]

The paper is structured as follows. Section 2 motivates by means of a scenario the interest of generic solvers for algebraic systems. Section 3 describes the basic algorithms and data structures used in FPsolve . Finally, Section 4 briefly describes the implementation.

2 Scenario: A Recommendation System

We succinctly describe SDSIRec, a recommendation system inspired by the SDSI authorization system [12], and very close to the reputation system described in [5].

SDSIRec distinguishes *customers* (denoted by x, y, z) and *products* (denoted by p). Given a collection of individual recommendations of products by customers, SDSIRec computes an aggregated customer rating for each product. Individual recommendations are described in SDSIRec by means of *rules* of the form:

$$x.\mathsf{Rec} \xrightarrow{w} p \tag{1}$$

$$x.\mathsf{Trust} \xrightarrow{w} y \tag{2}$$

The term $x.\mathsf{Rec}$ denotes the fuzzy set of all products recommended by customer x. The rule $x.\mathsf{Rec} \xrightarrow{w} p$ denotes that p belongs to $x.\mathsf{Rec}$ with weight w, i.e., that x recommends p with "rating" w. Analogously, $x.\mathsf{Trust}$ denotes the fuzzy set of all customers (whose recommendations are) trusted by x. The set of all weights, denoted by S, contains the special weight $\mathbb{0}$, which explicitly states that p resp. y does not belong to $x.\mathsf{Rec}$ resp. $x.\mathsf{Trust}$; assigning a rule the weight $\mathbb{0}$ is equivalent to removing the rule from the input.

Besides direct recommendation and direct trust, SDSIRec also takes into account indirect recommendation of products via trust in other customers. For instance, consider the following scenario:

$$\textsc{Jesse}.\mathsf{Trust} \xrightarrow{w_1} \textsc{Walt}$$

$$\textsc{Walt}.\mathsf{Rec} \xrightarrow{w_2} \text{FP}\textsc{solve}$$

Since Walt recommends FPsolve with weight w_2, and Jesse trusts the recommendations of Walt (his former high-school teacher) with weight w_1, SDSIRec infers that JESSE indirectly recommends FPSOLVE with some weight $w_1 \odot w_2$, where $\odot$ abstracts from the concrete way how the weights should be combined into a new weight. The operator must satisfy $\mathbb{0} \odot w = \mathbb{0} = w \odot \mathbb{0}$, so that the interpretation of $\mathbb{0}$ as a non-existing rule is preserved. The inference is modeled be the following (hard coded) rules:

$$x.\mathsf{Trust} \xrightarrow{\lambda} x.\mathsf{Trust}.\mathsf{Trust} \tag{3}$$
$$x.\mathsf{Rec} \xrightarrow{\mu} x.\mathsf{Trust}.\mathsf{Rec} \tag{4}$$

Rule (3) states that the set of customers trusted by x contains the set of customers trusted by customers trusted by x. Analogously, rule (4) states that the set of products recommended by x contains all products recommended by customers trusted by$x$. As these rules may lead to cycles, i.e. x might trust herself, thereby recommending to herself the products recommended by her, SDSIREC allows one to specify discount factors λ and μ to dampen resp. penalize these effects. The special weight $\mathbb{1}$ (required to satisfy $w \odot \mathbb{1} = w = \mathbb{1} \odot w$) can be used to disable this discounting. SDSIREC then treats the rules (1) to (4) as rewrite rules in the sense of a pushdown system [16]. For instance, we get

$$\text{JESSE}.\mathsf{Rec} \xrightarrow{\mu} \text{JESSE}.\mathsf{Trust}.\mathsf{Rec} \xrightarrow{w_1} \text{WALT}.\mathsf{Rec} \xrightarrow{w_2} \text{FPSOLVE}$$
$$\text{JESSE}.\mathsf{Rec} \xrightarrow{\mu} \text{JESSE}.\mathsf{Trust}.\mathsf{Rec} \xrightarrow{\lambda} \text{JESSE}.\mathsf{Trust}.\mathsf{Trust}.\mathsf{Rec} \xrightarrow{w_1} \text{WALT}.\mathsf{Trust}.\mathsf{Rec}$$

The first "path" with weight $\mu \odot w_1 \odot w_2$ captures that JESSE indirectly recommends FPSOLVE. The second path is an example of a path that cannot be extended to a recommendation of p: Since WALT trusts nobody (as specified by the input system), SDSIREC can never rewrite WALT.Trust to WALT.

In order to compute to what extent p belongs to $x.\mathsf{Rec}$ SDSIREC finally aggregates the weight of the (possibly infinitely many) paths leading from $x.\mathsf{Rec}$ to p. We use $\oplus$ to denote the operator that is used to aggregate the weights of different paths. It is well-known that if $\langle S, \oplus, \odot, \mathbb{0}, \mathbb{1} \rangle$ forms an *ω-continuous semiring*, then the problem of aggregating over all possible paths can be recast as computing the least solution of an algebraic system (see below) [9,16,6]. Recall that $\langle S, \oplus, \odot, \mathbb{0}, \mathbb{1} \rangle$ is a semiring if $\oplus$ and $\odot$ are associative and have neutral elements $\mathbb{0}$ and $\mathbb{1}$, respectively, $\oplus$ is commutative, $\odot$ distributes over $\oplus$, and any product with $\mathbb{0}$ as factor evaluates to $\mathbb{0}$. Given $a, b \in S$, we say $a \sqsubseteq b$ if there is $c \in S$ such that $a + c = b$. A semiring is *naturally ordered* if the relation $\sqsubseteq$ is a partial order. An *ω-continuous* semiring is a naturally ordered semiring extended by an infinite summation-operator $\sum$ that satisfies some natural properties. In particular, for every sequence $(a_i)_{i \geq 0}$ the supremum $\sup\{\sum_{0 \leq i \leq k} a_i \mid k \in \mathbb{N}\}$ w.r.t. $\sqsubseteq$ exists, and is equal to $\sum_{i \in \mathbb{N}} a_i$ [14].

Let R_{xp} and T_{xy} be variables standing for the total weights with which x recommends p or trusts y, and let r_{xp}, t_{xy} denote the weights of the direct recommendation, or the direct trust of x in p and y, respectively (i.e. $x.\mathsf{Rec} \xrightarrow{r_{xp}} p$ resp. $x.\mathsf{Trust} \xrightarrow{t_{xy}} y$).

If $\langle S, \oplus, \odot, \mathbb{0}, \mathbb{1}\rangle$ is an ω-continuous semiring, then the total weights are the unique smallest solution w.r.t. $\sqsubseteq$ of the following algebraic system (cf. [9,16,6])

$$R_{xp} = r_{xp} \oplus \bigoplus_y \mu \odot T_{xy} \odot R_{yp} \qquad \text{for all consumers } x, \text{ products } p$$

$$T_{xy} = t_{xy} \oplus \bigoplus_z \lambda \odot T_{xz} \odot T_{zy} \qquad \text{for all consumers } x, y$$

The key point of our argumentation is that in an application like the above we are interested in solving the same set of equations over many different semirings. Even further, users of the system may be interested in first defining their own semiring, and then solving the system. To illustrate this, let us examine several different interpretations of "weight", all of them very natural:

Weights as scores. The most natural interpretation of weights is perhaps as scores. A consumer x gives a product p or another consumer Y a score, corresponding to its degree of satisfaction with p, or its degree of trust in the recommendations of Y. If we assume that scores are real numbers in the interval $[0, 1]$, and choose $\oplus$ and $\otimes$ as sum and product of real numbers, we obtain the probabilistic semiring. Then R_{xp} represents the total weight of all recommendation paths leading from x to p. If we choose the Viterbi semiring $\langle [0, 1], \max, \cdot, 0, 1\rangle$ instead, then R_{xp} returns the weight of the strongest recommendation path.

Weights as expire times. Direct recommendations and trust, represented by rules of types (1) and (2,) can (and should) have an expire time. If we choose $\oplus$ to be the maximum and $\odot$ the minimum over the reals, then R_{xp} returns the earliest time at which all recommendation paths from x to p will have expired.

Weights as provenance information. If a system user does not trust some consumers, she may wish to compute, for each recommendation path from x to p, the set of consumers in the path. Or she may want to know the set of consumers visited along the recommendation path of maximal weight. Such provenance information can be computed within the semiring framework. For this it is convenient to treat all non-zero parameters $r_{xp}, t_{xy}, \lambda, \mu$ as formal parameters (free variables).

– To compute for each path the set of consumers involved, one can use the Why-semiring, well-known in provenance theory. Semiring elements are sets of sets of consumers. We set $r_{xp} = \{\{x\}\}$ and $t_{xy} = \{\{x, y\}\}$ (and treat λ as $\{\{\lambda\}\}$ and μ as $\{\{\mu\}\}$), and define:
  - $\{X_1, \ldots, X_n\} \odot \{Y_1, \ldots, Y_n\} := \{X_1 \cup Y_1, x_1 \cup Y_2, \ldots, X_n \cup Y_m\}$, and
  - $\{X_1, \ldots, X_n\} \oplus \{Y_1, \ldots, Y_n\} := \{X_1, \ldots, X_n, Y_1, \ldots, Y_m\}$.
– If we wish to compute the provenance of the recommendation of maximal weight, we can use the following semiring: as semiring elements we choose the pairs (α, X), where $\alpha \in [0, 1]$ and X is a set of consumers. We set $t_{xy} = (w, \{x, y\})$ with $w \in (0, 1]$, and analogously for r_{xp}. The abstract operators are instantiated as follows:

- $(\alpha_1, X_1) \odot (\alpha_2, X_2) := (\alpha_1\alpha_2, X_1 \cup X_2)$, and
- $(\alpha_1, X_1) \oplus (\alpha_2, X_2) := (\max\{\alpha_1, \alpha_2\}, \textbf{if } \alpha_1 \geq \alpha_2 \textbf{ then } X_1 \textbf{ else } X_2)$

These examples show that, instead of creating new tools for each new semiring, it can be better to implement a generic tool, with generic algorithms applicable to any semiring, or at least to any semiring in a broad class.

3 Algorithms and Data Structures

The two main generic schemes implemented in FPsolve for the approximation (and sometimes exact computation) of the least solution of an algebraic system are classical fixpoint iteration and Newton's method. Following [15,10], we introduce them as procedures that "unfold" the algebraic system up to a certain depth which allows both to unify and at the same time simplify their presentation.

Classical fixed-point iteration. Given an algebraic system of equations $X = F(X)$ over an ω-continuous semiring, Kleene's theorem states that the system has a unique least solution μF with respect to the natural order $\sqsubseteq$, and that μF is the supremum of the sequence $F(0), F^2(0), \ldots, F^i(0)$ [14]. So μF can be approximated by computing successive elements of the sequence. If the semiring further satisfies the *ascending chain property* (for every ω-chain $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ eventually $a_k = a_{k+1} = a_{k+2} = \ldots$) then $\mu F = F^i(0)$ for some $i \geq 0$, and so μF can be effectively computed.

As explained in e.g. [15], an algebraic system can be associated a context-free grammar. For instance, for the system

$$X = a \odot X \odot X \oplus b \tag{5}$$

we obtain the grammar

$$X \rightarrow aXX \mid b \tag{6}$$

Conversely, we assign to a derivation tree of the grammar a value in the semiring, given by the product of its leaves (the ordered product if the semiring is not commutative); further, we assign to a set of derivation trees the sum of the values of its elements. The following result can be proved by a simple induction on k:

> $F^k(0)$ is the sum over the set of all derivation trees of the grammar of height less than k.

Building on this observation, for every k we "unfold" (5) into an acyclic system over variables $X^{<h}$ and $X^{=h}$ for every $h \leq k$, such that the solutions of $X^{<h}$ and $X^{=h}$ are the values of the derivation trees of height *less* than h and *equal* to h, respectively. For this, we obviously have to set

$$X^{<0} = 0 \quad X^{=0} = b \quad \text{ and } \quad X^{<h+1} = X^{<h} \oplus X^{=h} \text{ for all } h \in \mathbb{N} \tag{7}$$

In order to obtain the defining equation of $X^{=h}$ we observe that the trees of height h can be partitioned into those whose left subtree has height $h-1$, and those whose left subtree has height strictly smaller than $h-1$ *and* whose right subtree has height $h-1$, i.e. we partition by means of the first position from the right at which a subtree of height exactly $h-1$ is rooted. This leads to

$$X^{=h} = a \odot X^{<h} \odot X^{=h-1} \oplus a \odot X^{=h-1} \odot X^{<h-1}. \tag{8}$$

We can see the unfolding up to depth k as a symbolic representation of $F^k(0)$, which implicitly uses subterm sharing (*arithmetic circuit*). When the coefficients of the algebraic system (a, b in our example) are formal parameters, we can efficiently compute $F^k(0)$ for different values a, b, by just plugging them into the unfolded system.

Newton's method. Newton's method for arbitrary ω-continuous semirings, as described in [9], can be much faster than Kleene iteration. It is shown in [10] that the method can also be presented as an unfolding of the algebraic system: This time, the system is unfolded w.r.t. the *Strahler number* or *dimension* of its associated derivation trees (see [10,15]). The dimension of a rooted tree t is defined as the height of the largest perfect binary tree that is a minor of t.

Consider again equation (5). We split X into a family of variables $X^{<d}$ and $X^{=d}$ for $d \in \mathbb{N}$. The solutions of the unfolded system for $X^{<d}$ and $X^{=d}$ will now be the value of the derivation trees of dimension *less* than d, and *equal* to d, respectively. Just as before, we have

$$X^{<0} = 0 \quad X^{=0} = b \quad \text{and} \quad X^{<d+1} = X^{<d} \oplus X^{=d} \text{ for all } d \in \mathbb{N}. \tag{9}$$

In order to derive the defining equation of $X^{=d}$, observe that there are three possible cases for a tree of dimension d: either the left subtree has dimension d, and the right subtree has dimension at most $d-1$; or vice versa; or both subtrees have dimension exactly $d-1$ (this is the case in which the root of the minor coincides with the root of the tree). So we get

$$X^{=d} = a \odot X^{=d} \odot X^{<d} \oplus a \odot X^{<d} \odot X^{=d} \oplus a \odot X^{=d-1} \odot X^{=d-1}. \tag{10}$$

However, this unfolding does not represent an arithmetic circuit as it is not yet acyclic: $X^{=d}$ appears on both sides of the equation. But equation (10) is linear in $X^{=d}$, and so, if multiplication is commutative, we can replace it by (with $\mathbb{1} \oplus \mathbb{1} = \mathbb{2}$)

$$X^{=d} = \mathbb{2} \odot a \odot X^{=d} \odot X^{<d} \oplus a \odot X^{=d-1} \odot X^{=d-1} \tag{11}$$

and use Kleene's theorem [14] to replace it by

$$X^{=d} = \left(\mathbb{2} \odot a X^{<d}\right)^{*} \odot a \odot \left(X^{=d-1}\right)^{2} \tag{12}$$

where the Kleene star is defined, as usual, by $x^* := \sum_{k \in \mathbb{N}} x^k$ (and is well defined for any ω-continuous semiring).[2] The new system is acyclic, i.e. an arithmetic

[2] In the noncommutative case, one may resort to an instance of the semiring of contexts in order to obtain a rational tree expression.

circuit w.r.t. $\oplus$, $\odot$, and $*$, and as in the previous case, can be used as a compact symbolic representation of the d-th Newton approximation, useful when a, b are formal parameters (see Fig. 1 for an example). In particular, every Newton approximation can be represented by means of a rational expression.

To actually compute the solution for particular values of a and b, we can then use straight-forward constant propagation going from bottom ($X^{=0}, X^{<0}$) to top ($X^{<d}$). However, for this the Kleene star x^* must be effectively computable in the given semiring representation. This is indeed the case for several important semirings. The simplest example is the probability semiring, where for every rational number $x \in [0, 1)$ we have $x^* = 1/(1 - x)$. Tropical semirings are another example. For instance, over the integers extended by least ($-\infty$) and greatest element ($+\infty$) , with addition given by min and multiplication given by + (on $\mathbb{Z}$), we have $x^* = 0$ if $x \geq 0$ and $x = -\infty$ otherwise. A third example is the semiring of semilinear sets of vectors with components in $\mathbb{N} \cup \{\infty\}$, with $X \oplus Y := X \cup Y$, and $X \odot Y := \{x + y \mid x \in X, y \in Y\}$.

Connection to Newton's method over the reals. Applying Newton's method to $g(X) := f(X) - X = aX^2 - X + b$ (interpreted over the reals) starting form the initial approximation $X = 0$ we obtain the sequence:

$$X_0 := 0 \quad X_{d+1} = X_d - \frac{g(X_d)}{g'(X_d)} = X_d - \frac{aX_d^2 - X_d + b}{2aX_d - 1} = X_d + \frac{aX_d^2 - X_d + b}{1 - 2aX_d}.$$

Setting $Y_d := X_{d+1} - X_d$ this can be written as

$$Y_d = 2aX_dY_d + (aX_d^2 - X_d + b).$$

Straight-forward induction now shows that over the nonnegative reals the values of X_d and $X^{<d}$ resp. Y_d and $X^{=d}$ coincide [10]. In particular, the defining equation of $X^{=d}$ can be seen as the generalization of the derivative of aX^2 in the noncommutative case.

Multivariate case. Both unfoldings immediately generalize to the setting of multiple variables $X, Y, \ldots$. As mentioned above, in the univariate case we use the fact that the solution of an equation $X = aX + b$ is given by a^*b. In the multivariate case, when the semiring is commutative, we have to deal with systems of linear equations $X = AX + B$ for a matrix A and a vector B over the semiring. It is well known that the solution is given by A^*B, where $A^* = \sum_{i=0}^{\infty} A^i$, and matrix multiplication is defined as for the natural or the real numbers, but replacing sum and product by the operations of the semiring being considered. In the next section, we describe the two algorithms for computing A^* implemented in FPsolve.

3.1 Solving Linear Equations

As mentioned above, solving a linear equation amounts to computing A^* for a given square matrix A over a semiring. Given a matrix A, the algorithms returns a matrix whose elements are semiring expressions over the semiring operations and the Kleene star. So, intuitively, the algorithms reduce the problem of computing the star of a matrix to computing the star of semiring elements.

FPsolve implements both the well-known Floyd-Warshall algorithm, and the recursive divide-and-conquer approach.

Generalized Floyd-Warshall The Floyd-Warshall algorithm for solving the all-pairs-shortest-path problem in weighted (finite) graphs carries directly over to the setting of generic ω-continuous semirings, even if addition $\oplus$ is not idempotent (cf. [7]). (In fact, it suffices when the semiring $\langle S, \oplus, \odot, \mathbb{0}, \mathbb{1}\rangle$ is closed but not necessarily ω-continuous.) The following description is an optimized variant of the algorithm in [7] which reduces the number of semiring operations required.

```
input  : Matrix $\mathbf{A} \in S^{n\times n}$ over a semiring $S$.
output: Reflexive-transitive closure $\mathbf{A}^*$.

$B := A$
for $k = 1 \dots n$ do
    $B_{k,k} := B^*_{k,k}$
    for $i = 1 \dots n,\ i \neq k$ do
        $B_{i,k} = B_{i,k} \odot B_{k,k}$
        for $j = 1 \dots n,\ j \neq k$ do
            $B_{i,j} := B_{i,j} \oplus B_{i,k} \odot B_{k,j}$
        end
    end
    for $j = 1 \dots n, j \neq k$ do
        $B_{k,j} = B_{k,k} \odot B_{k,j}$
    end
end
return $B$
```

Algorithm 1: Generalized Floyd-Warshall algorithm over semirings.

From the description of the algorithm it is easy to count that the total number of semiring operations (i.e. $+, \cdot, ^*$) needed is $T(n) = 2n^3 - 2n^2 + n \in \Theta(n^3)$.

Divide-and-conquer This algorithm recursively applies the formula for computing the Kleene star of a 2×2-matrix:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \qquad \mathbf{M}^* = \begin{bmatrix} \mathbf{F} & \boldsymbol{\alpha}\mathbf{G}^* \\ \mathbf{G}^*\boldsymbol{\beta} & \mathbf{G}^* \end{bmatrix} \qquad \text{with} \begin{array}{l} \boldsymbol{\alpha} = \mathbf{A}^*\mathbf{B} \\ \boldsymbol{\beta} = \mathbf{C}\mathbf{A}^* \\ \mathbf{G} = \mathbf{D} + \mathbf{C}\boldsymbol{\alpha} \\ \mathbf{F} = \boldsymbol{\alpha}\mathbf{G}^*\boldsymbol{\beta} + \mathbf{A}^* \end{array} .$$

Given a $n \times n$-matrix $\mathbf{M}$ ($n > 2$), the entries $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ become submatrices of $\mathbf{M}$ to which the algorithm is then applied recursively; the recursion stops when either $n = 2$ or $n = 1$. A formal proof of correctness (for any *Conway* semiring) goes back to Ésik and Kuich (cf. [7]).

Altogether we need two recursive calls, six matrix multiplications (the term $\boldsymbol{\alpha}\mathbf{G}^*$ appears twice and thus needs to be evaluated only once), and two matrix additions. Hence, the number of operations needed by this algorithm can be expressed by the recurrence relation [3]

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 6\left[2\left(\frac{n}{2}\right)^3 - \left(\frac{n}{2}\right)^2\right] + 2\left(\frac{n}{2}\right)^2 \\ &= 2T\left(\frac{n}{2}\right) + \frac{3}{2}n^3 - n^2 \end{aligned}$$

Which can be solved exactly (setting $T(n) = 1$ and $n = 2^l$) resulting in $T(n) = 2n^3 - 2n^2 + n \in \Theta(n^3)$. Hence this algorithm uses the same number of operations as Floyd-Warshall. Both algorithms need $n^3 - 2n^2 + n$ additions, $n^3 - n$ multiplications, and n Kleene stars.

Symbolic solving Recall our initial example

$$X = a \odot X \odot X \oplus b$$

and its unfolding w.r.t. dimension for an arbitrary $d \in \mathbb{N}$ (assuming $\odot$ is commutative)

$$X^{=d} = 2 \odot a \odot X^{=d} \odot X^{<d} \oplus a \odot X^{=d-1} \odot X^{=d-1}$$

As $X^{<d} = X^{<d-1} \oplus X^{=d-1}$, every iteration of Newton's method essentially consists of solving this linear equation after substituting for the variables $X^{<d}$ and $X^{=d-1}$ the already computed solution. Analogously, in the multivariate setting essentially the same linear equation system has to be solved over and over again. FPsolve thus allows to first compute a symbolic solution of the linear system by treating $X^{<d}$ and $X^{=d-1}$ as formal parameters which allows to share common subexpressions and thus obtain a succinct symbolic representation of a Newton approximation. This allows to efficiently evaluate a Newton approximation of an algebraic system for several different semiring interpretations.

Consider the generic linear equation system

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

Treating $a, \ldots, f$ as formal parameters over some semiring, the (symbolic) solution of this system is given by

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a^*b(ca^*b \oplus d)^*ca^*e \oplus a^*b(ca^*b \oplus d)^*f \oplus a^*e \\ (ca^*b \oplus d)^*ca^*e \oplus (ca^*b \oplus d)^*f \end{pmatrix}$$

where we have omitted the $\odot$ for readability.

[3] Note that multiplying two $n \times n$ matrices requires $n^3 - n^2$ operations (via the schoolbook method – we cannot use e.g. Strassen's algorithm as we lack a difference operator!).

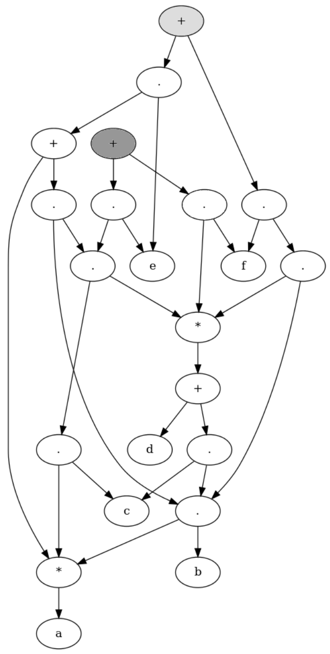

Fig. 1. Succinctly representing all terms of the product $A^* \cdot (e, f)^T$ via a BDD-like sharing of subexpressions. By reversing the direction of all edges this can be read as an arithmetic circuit with output gates colored in grey.

The internal representation of these terms is shown in Fig. 1: FPsolve stores the expressions as part of an "abstract syntax DAG" (reversing the direction of the edges we obtain an arithmetic circuit with gates for addition, multiplication, and Kleene star) similar to BDD libraries like CUDD, where we have colored the x- resp. y-component in light resp. dark grey; this representation allows to reduce both the memory consumption, and the reevaluation of identical subterms.

In this simple 2×2 case the concrete recursive approach (as stated above) computes 10 semiring operations, the same if the symbolic solution is computed (using the same recursive algorithm). This holds in general if all elements of the input matrix are different. However, in general input matrices can have the same element in many different positions, then even the recursive algorithm will compute some identical subexpressions multiple times (that occur in different execution branches) since it cannot guess them a priori. In this case, symbolic solving allows for a *global* subexpression detection after the whole matrix-star has been computed.

Although the symbolic approach significantly reduces the number of semiring operations needed, the overhead from computing and storing the symbolic solution is not always negligible. This is particularly true for numeric semirings (like the semiring of positive reals) that are implemented using machine precision floating point numbers – for these the semiring operations are so fast that the overhead outweighs the benefits of symbolic solving.

We therefore give the user the freedom of choice whether to use the concrete (i.e. in every iteration) or symbolic (i.e. solve once then plug in in every iteration) method of solving linear equations.

3.2 Decomposition into strongly-connected components

To efficiently process large algebraic systems, FPsolve supports a decomposition of the system into strongly connected components (SCCs). To make this precise recall the definition of *dependency graph*: Its nodes are the variables occurring in the algebraic system; its edges are induced by the defining equations: we have an edge from variable X to variable Y if Y occurs in the defining equation of X. X *depends* on Y if there is a path from X to Y in the dependency graph. To determine the value of variable X it then suffices to determine the values of all variables on which X depends. Using Tarjan's algorithm we therefore partition the dependency graph into SCCs, and process these SCCs in reverse topological ordering ("bottom up"). In particular when using Newton's method this can lead to a noticeable speed-up in the computation of the Kleene star.

4 Implementation

Currently, FPsolve comprises roughly 8,000 lines of C++. The code can be obtained freely from `https://github.com/mschlund/FPsolve`. We use several existing frameworks and libraries:

- CPPUnit for writing unit-tests.
- boost for IO-tasks (parsing, command-line arguments).
- Genepi, Mona, and Lash for representing semilinear sets via NDDs.
- libfa for representing elements of "lossy" semirings (i.e. semirings satisfying $1 \sqsubseteq a$ for any semiring element $a \neq \mathbb{0}$ – this generalizes the downward closure of languages) as finite automata.

FPsolve features data structures for commutative as well as non-commutative polynomials, different solvers (semi-naive fixpoint iteration, Newton's method), and several predefined semirings (semilinear sets, real numbers, tropical and boolean semiring) as well as some generic constructions (via C++ templates) to build new semirings from existing ones like the direct product of two semirings or the semiring of matrices over some semiring.

The focus of our library is to provide generic algorithms and to be easily extensible. One of our goals was to make it easy for users to write their own semiring-constructions or tailor the generic solving algorithms to their needs.

The library consists of three main parts:

- Data structures (polynomials, matrices, BDD-like DAG-structure to support subterm sharing)
- Semirings (semilinear sets, positive real numbers, why semiring, generic product semiring, ...)
- Solvers (Kleene solver, Newton solver)

Figure 2 shows a simplified view of the main structure of our library. Observe that many classes are templated which produces efficient code due to compile-time polymorphism.

4.1 Invocation of the Standalone Solver

FPsolve also includes a callable solver and a parser for equation systems that demonstrates the use of the library.

To apply the standalone solver, one has to describe the algebraic system as a BNF-style context-free grammar. Variables of the system are enclosed in angle brackets, multiplication is not explicitly written, the addition $x + y$ is written as $x \mid y$. To solve the following system over the reals

$$X = 0.5XY + 0.5 \qquad Y = 0.3Y + 0.7X$$

we would create a text file `test.g` containing:

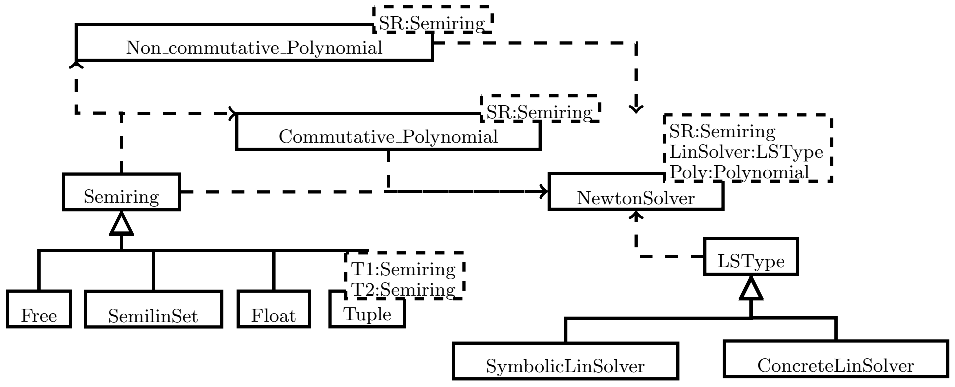

Fig. 2. A (simplified) part of our architecture.

```
<X> ::= 0.5 <X> <Y>  | 0.5;
<Y> ::= 0.3 <Y> | 0.7 <X>;
```

The simplest invocation of the tool is then

```
$ ./fpsolve -f test.g --float
```

This minimal set of parameters specifies

1. the input file containing the algebraic system (`-f test.g`).
2. the semiring over which the system and its constants (like `0.3`) are to be interpreted (here `--float`).

The tool outputs:

```
$ ./fpsolve -f test.g --float
Newton Concrete
Iterations: 3
Solving time: 0 ms (196 us)
X == 0.875
Y == 0.875
```

By default, the number of Newton iterations for a system of n equations is $n + 1$ – for commutative, idempotent semirings this suffices to compute the exact solution [8].

A more sophisticated use of the tool's options would be the following:

```
$ ./fpsolve -f test.g --float -i 10 -s newtonSymb
Newton Symbolic
Iterations: 10
Solving time: 0 ms (536 us)
X == 0.999023
Y == 0.999023
```

Here, we select

1. the number of iterations (`-i 10`)
2. the solving algorithm to use (switch `-s`), possible choices are `newtonSymb`, `newtonConc`, `kleene`.

For larger equation systems there is the possibility to decompose the system into SCCs and solve them bottom-up (switch `--scc`).

4.2 Custom Semirings and Extensions

It is very easy and straightforward to extend our library with new semirings, it merely requires three steps:

- Implement all semiring operations (addition $\oplus$, multiplication $\odot$, star $*$)
- Define a constructor that takes a string-argument (effectively a small parser)
- Add a new command-line switch to the main-method together with a call to the solving function.

The second point delegates the IO/parsing task for semiring-elements to the implementer. This enables us to parse equation systems into the most general intermediate format (non-commutative polynomials over the free semiring) and then to map these to the user-defined semiring. Since our input-parser takes quite some time to compile (due to `boost::spirit` and templates), by this approach we avoid to touch the parser and the need for recompilation.

The semiring operations $\odot, \oplus$ (`+` and `*`) are implemented in the abstract base-class `Semiring` using `+=` and `*=`. Any new semiring should be derived from the abstract class `StarableSemiring` and has to implement the three operations `*=`, `+=`,`star()`. Take for instance the "MaxProvenance" semiring from the end of Section 2 consisting of pairs (α, X) of real numbers and sets of variables with

$$\begin{aligned}(\alpha_1, X_1) \odot (\alpha_2, X_2) &:= (\alpha_1\alpha_2, X_1 \cup X_2)\\ (\alpha_1, X_1) \oplus (\alpha_2, X_2) &:= (\max\{\alpha_1, \alpha_2\}, \textbf{if } \alpha_1 \geq \alpha_2 \textbf{ then } X_1 \textbf{ else } X_2)\\ (\alpha, X)^* &:= (1, \emptyset)\end{aligned}$$

To implement this simple semiring, we derive from `StarableSemiring` the new class `MaxProvSR` with members `weight` and `prov` storing α (e.g. as a `float`) and X (e.g. as a `set<>`), respectively. What remains is then to implement the three operators `*=`, `+=`,`star()`. For instance, the addition-assignment operator could be implemented as

```
MaxProvSR MaxProvSR::operator+=(const MaxProvSR& elem)
{
  if(this->weight < elem.weight) {
    this->weight = elem.weight;
    this->prov = elem.prov;
  }
  return *this;
}
```

Inheritance then takes care of the implementation of the addition operator. Implementing the remaining two operators is just as straight-forward. To make the semiring available in the command line tool, a corresponding switch and a parser for reading semiring elements from the input have to implemented in addition.

To check our claim of "easy extendability", we made a rather naive implementation of the Why-semiring for this paper which took about two hours (until all bugs were eliminated[4]). Once a new semiring is defined and the main-method is adapted, all solvers just work out-of-the-box to solve algebraic systems like the following (file `test/grammars/bintrees.g`):

```
<X> ::= a<X><X> | c;
```

```
$ ./fpsolve --why -f ../test/grammars/bintrees.g
Newton Concrete
Iterations: 2
Solving time: 0 ms (214 us)
X == {{a,c},{c}}
```

```
$ ./fpsolve --why -f ../test/grammars/bintrees.g -s kleene -i 2
Kleene solver
Iterations: 2
Solving time: 0 ms (281 us)
X == {{a,c},{c}}
```

5 Conclusions and Related Tools

We have introduced FPsolve, an implementation of generic algorithms for solving fixpoint equations on semirings. The algorithms are parametric on the semiring. New semirings can be easily added by defining implementations of the sum, product and (possibly) Kleene star operations.

As mentioned in the introduction, many program analysis problems can be reduced to solving fixpoint equations on semirings. This has lead to a number of implementations and tools. An early effort is the Fixpoint-Analysis Machine for solving systems of boolean fixpoint equations [17]. The tool can deal with hierarchical and alternating fixpoints, but is not parametric on the equation domain. The Weighted Pushdown Systems Library and Weighted Automata Library (see [16,3,2]), and Goblint (see [4,1] implement many sophisticated algorithms for semirings satisfying the ascending chain condition.

While FPsolve is currently an academic tool, we have illustrated its potential interest outside theoretical computer science by means of an application scenario, namely a recommendation system. Genericity allows the users of the system to aggregate the information given by individual recommendations in different, personalized ways, by defining their own semiring.

[4] We developed a small collection of unit-tests (also generic tests that can be instantiated with any semiring) and encourage any user who implements new semirings to use and adapt them during development.

Acknowledgments We thank Michael Kerscher and Michał Terepeta for their help in developing and extending FPsolve .

References

1. Goblint, `http://goblint.in.tum.de/`
2. WALi: the Weighted Automata Library, `https://research.cs.wisc.edu/wpis/wpds/`
3. Weighted Pushdown Systems Library, `http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/wpds/`
4. Apinis, K., Seidl, H., Vojdani, V.: How to combine widening and narrowing for non-monotonic systems of equations. In: Boehm, H.J., Flanagan, C. (eds.) PLDI. pp. 377–386. ACM (2013)
5. Bouajjani, A., Esparza, J., Schwoon, S., Suwimonteerabuth, D.: SDSIrep: A Reputation System Based on SDSI. In: TACAS. pp. 501–516 (2008)
6. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551– (2003)
7. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer (2009)
8. Esparza, J., Kiefer, S., Luttenberger, M.: On Fixed Point Equations over Commutative Semirings. In: STACS. pp. 296–307 (2007)
9. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian Program Analysis. J. ACM 57(6), 33 (2010)
10. Esparza, J., Luttenberger, M.: Solving fixed-point equations by derivation tree analysis. In: CALCO. pp. 19–35 (2011)
11. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS. pp. 31–40 (2007)
12. Jha, S., Reps, T.W.: Model checking SPKI/SDSI. Journal of Computer Security 12(3-4), 317–353 (2004)
13. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: CC. pp. 125–140 (1992)
14. Kuich, W.: Handbook of Formal Languages, vol. 1, chap. 9: Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata, pp. 609 – 677. Springer (1997)
15. Luttenberger, M., Schlund, M.: Convergence of Newton's Method over Commutative Semirings. In: LATA. LNCS, vol. 7810, pp. 407–418 (2013)
16. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58(1–2), 206–263 (October 2005), special Issue on the Static Analysis Symposium 2003
17. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: Lee, I., Smolka, S.A. (eds.) CONCUR. Lecture Notes in Computer Science, vol. 962, pp. 72–87. Springer (1995)