Munchausen Iteration

Roland Meyer and Sebastian Muskalla

TU Kaiserslautern, {meyer, muskalla}@cs.uni-kl.de

Abstract

We present a method for solving polynomial equations over idempotent ω-continuous semirings. The idea is to iterate over the semiring of functions rather than the semiring of interest, and only evaluate when needed. The key operation is substitution. In the initial step, we compute a linear completion of the system of equations that exhaustively inserts the equations into one another. With functions as approximants, the following steps insert the current approximant into itself. Since the iteration improves its precision by substitution rather than computation we named it Munchausen, after the fictional baron that pulled himself out of a swamp by his own hair. The first result shows that an evaluation of the n^{th} Munchausen approximant coincides with the $2^{n\text{th}}$ Newton approximant. Second, we show how to compute linear completions with standard techniques from automata theory. In particular, we are not bound to (but can use) the notion of differentials prominent in Newton iteration.

1 Introduction

Verification problems pop up in a fascinating variety of applications. Despite this variety, they are often formulated in a uniform way, as finding the least solution to a system of polynomial equations that is interpreted over a semiring. The verification tasks that can be captured this way range from language-theoretic problems underlying model checking [7] to dataflow analyses [10] needed in compilers [16]. The programs that can be handled may involve recursion [17, 14] and weak forms of parallelism [13, 8]. Technically, the system of equations captures the flow of control in the program of interest. The semiring interpretation models the aspects of the program semantics that influence the verification task. The least solution is the most precise semantic information (of the form one has chosen to track) that is invariant under the program commands.

Computing the least solution to a given system of equations is an algorithmic challenge, commonly referred to as state space explosion. The least solution is the least fixed point of the right-hand side functions. The predominant method for computing this fixed point is Kleene iteration. In its plain form, Kleene iteration understands the right-hand side functions as a single function over the product domain and applies it over and over again until the least fixed point is reached. The practical importance of Kleene iteration stems from the fact that it is amenable to algorithmic optimizations. In particular, rather than working on the product domain, implementations use a worklist that only stores the functions whose variables have received updates [1, 11].

Recently, Esparza et al. proposed Newton iteration [3], a new method for computing least fixed points that combines Kleene iteration with an acceleration principle (see also [9, 6] for precursors of the method). The idea is indeed inspired by the method for finding roots of numerical functions. The current approximant is not only modified by an application of the function, like in Kleene iteration, but in addition shifted towards the fixed point by an acceleration that makes use of the function's differential. Newton iteration is guaranteed to converge to the least fixed point, and to do so faster than Kleene iteration (there are even cases where Newton reaches the fixed point while Kleene does not). On the downside, the Newton steps are computationally more expensive than Kleene steps. In particular, Newton iteration does not yet decompose into a worklist procedure. There has, however, been recent interest in the algorithmics of the method [5, 15].

Our contribution is a new iteration scheme for solving systems of polynomial equations $\mathbf{x} = \mathbf{f}(\mathbf{x}) + \mathbf{a}$ over semirings. To be precise, we work with idempotent and ω-continuous semirings, assumptions that are typically met in verification. Our iteration is exponentially faster than Newton and (arguably) easier to compute. To explain the idea, note that both, Kleene and Newton, compute in the semiring of interest. Our method works symbolically, over the semiring of functions. The key idea is substitution. In the initial step, we compute a so-called linear completion of the system of equations, $\beta^{(0)} = lc(\mathbf{f})$. The completion exhaustively inserts the right-hand side functions into each other. This means an occurrence of variable y in $\mathbf{f}_x$ is replaced by $\mathbf{f}_y$. For the resulting function to remain representable, we restrict ourselves to a completion process that is linear in the sense that the next replacement will be done on a variable in $\mathbf{f}_y$. In the iteration step, we continue to insert the current approximant into itself, $\beta^{(n+1)} = eval_{\beta^{(n)}}(\beta^{(n)})$. We obtain semantic and algorithmic results about Munchausen iteration.

Concerning the semantics, we show that the Munchausen sequence is faster than the Newton sequence: When we evaluate the n^{th} Munchausen approximant at the constant vector $\mathbf{a}$, we obtain the $2^{n^{\text{th}}}$ Newton approximant. This precise correspondence allows us to transfer deep results from [3]: The Munchausen sequence converges to the least fixed point and, in the commutative case, is guaranteed to reach the least fixed point in a number of steps that is logarithmic in the number of variables. As second main result, we show that there is some flexibility in where to evaluate the approximants. Any $\mathbf{b}$ chosen between $\mathbf{a}$ and the least fixed point will guarantee convergence to the least fixed point. Munchausen iteration thus combines well with further accelerations.

Concerning the algorithmics, we study the operations of linear completion $\beta^{(0)} = lc(\mathbf{f})$ and evaluation $\beta^{(n+1)} = eval_{\beta^{(n)}}(\beta^{(n)})$ We show that the linear completion of a function is a linear context-free language. Moreover, this language can be represented symbolically by a regular expression, provided the semiring has associated a suitable tensor operation. The idea of using tensors was introduced recently in the context of Newton iteration [15]. Our contribution is to lift it to the semiring of functions. As a second result, we show how to compute the evaluation on the symbolic representation of the linear completion (a linear context-free grammar or even a regular expression). The main finding is that the iteration steps are well-behaved to an extent that we can implement them by an indexed language.

Outline Section 2 introduces the basics on semirings. The Munchausen iteration scheme is presented in Section 3, together with the study of its semantic properties. The algorithmics of our iteration is the subject of Section 4. Section 5 concludes the paper with a discussion on the implementation of the method.

Proofs missing in the paper can be found in the appendix.

2 Systems of Polynomial Equations over ω-Continuous Semirings

We study systems of polynomial equations over idempotent and ω-continuous semirings. A **semiring** $\mathcal{S}$ is a tuple $(\mathcal{S}, \oplus, \odot, 0, 1)$ with the following properties (for all $a, b, c \in \mathcal{S}$):

$(\mathcal{S}, \oplus, 0)$ is a commutative monoid $\qquad$ $(\mathcal{S}, \odot, 1)$ is a monoid

$$a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c) \qquad (b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$$

$$a \odot 0 = 0 \odot a = 0 \;.$$

A semiring comes with the so-called **natural ordering** $\leq$ on its elements: $a \leq a \oplus b$ for all $a, b \in \mathcal{S}$. The semiring is called naturally ordered if $\leq$ is a partial order. In the following, we will determine suprema over sets of semiring elements. These suprema will always be taken wrt. the natural ordering.

A naturally ordered semiring is **ω-continuous** if it satisfies the following Properties (1) to (4). Property (1) requires chains $(a_i)_{i \in \mathbb{N}}$ to have a supremum $\sup\{a_i \mid i \in \mathbb{N}\}$ in $\mathcal{S}$. Recall that a chain is a sequence with $a_i \leq a_{i+1}$ for all $i \in \mathbb{N}$. To state the Properties (2) to (4), given a sequence $(a_i)_{i \in \mathbb{N}}$ in $\mathcal{S}$ we define the infinite sum

$$\bigoplus_{i \in \mathbb{N}} a_i \;=\; \sup\{a_0 \oplus \ldots \oplus a_i \mid i \in \mathbb{N}\} \;.$$

Note that the sum exists by Property (1). The Properties (2) to (4) now require

$$c \odot (\bigoplus_{i \in \mathbb{N}} a_i) = \bigoplus_{i \in \mathbb{N}} (c \odot a_i) \qquad (\bigoplus_{i \in \mathbb{N}} a_i) \odot c = \bigoplus_{i \in \mathbb{N}} (a_i \odot c) \qquad \bigoplus_{i \in \mathbb{N}} a_i = \bigoplus_{i \in I} \bigoplus_{j \in J_i} a_j \;.$$

The latter equality is supposed to hold for every partitioning of the natural numbers. The requirement for ω-continuity allows us to define the **Kleene-star** operator $^* : \mathcal{S} \to \mathcal{S}$ by $a^* = \bigoplus_{i \in \mathbb{N}} a^i$ where we set $a^0 = 1$.

Throughout, we will work with idempotent and ω-continuous semirings, **io-semirings** for short. A semiring is **idempotent** if addition is idempotent, $a \oplus a = a$ for all $a \in \mathcal{S}$. We will also consider the special case of **commutative** io-semirings, where besides addition also multiplication is commutative.

Given a finite set of variables $\mathcal{X} = \{x_1, \ldots, x_k\}$, a **monomial** is an expression of the form $m = a_1 \odot x_{i_1} \odot \ldots \odot a_l \odot x_{i_l} \odot a_{l+1}$, where all $a_i \in \mathcal{S}$ and $x_{i_j} \in \mathcal{X}$. A monomial without variables is a **constant**. A **polynomial** is a finite sum of monomials, $p = \bigoplus_{i=1,\ldots,k} m_i$. We use $\mathcal{S}[\mathcal{X}]$ for the set of all polynomials. A **power series** is a countable sum of monomials.

The set of functions over $\mathcal{S}$ with arguments $\mathcal{X}$, denoted by $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}$, forms a semiring with element-wise addition and multiplication, i.e. for all $\mathbf{a} \in \mathcal{S}^{\mathcal{X}}$:

$$(f \oplus g)(\mathbf{a}) = f(\mathbf{a}) \oplus g(\mathbf{a}) \qquad (f \odot g)(\mathbf{a}) = f(\mathbf{a}) \odot g(\mathbf{a}) \;.$$

The **semiring of functions** $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}$ is ω-continuous (idempotent, commutative) if and only if $\mathcal{S}$ is ω-continuous (idempotent, commutative). We will also consider $\mathcal{X}$-dimensional vectors of such functions, i.e. the set $(\mathcal{S}^{\mathcal{X}} \to \mathcal{S})^{\mathcal{X}}$. We can see such a vector as a single function $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$. Together with the component-wise operations, the set of functions $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$ again forms a semiring, and the properties of $\mathcal{S}$ carry over. We are particularly interested in functions defined by polynomials. Note that they are **monotone**.

Given a single function $f : \mathcal{S}^{\mathcal{X}} \to \mathcal{S}$ and a vector of functions $\mathbf{v} : \mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$, we write $eval_{\mathbf{v}}(f)$ for the composition of f and $\mathbf{v}$ defined by

$$eval_{\mathbf{v}}(f) : \mathcal{S}^{\mathcal{X}} \to \mathcal{S} \qquad eval_{\mathbf{v}}(f)(\mathbf{a}) = f(\mathbf{v}_{x_1}(\mathbf{a}), \ldots, \mathbf{v}_{x_k}(\mathbf{a})) \;.$$

We also use evaluation on vectors of functions, $(eval_{\mathbf{v}}(\mathbf{f}))_x = eval_{\mathbf{v}}(\mathbf{f}_x)$. If $\mathbf{a} \in \mathcal{S}^{\mathcal{X}}$ is a vector of values, we moreover let $eval_{\mathbf{a}}(f)$ denote the value of f at $\mathbf{a}$. Evaluation is monotone in both, the function and the argument, and it is associative. Let $\mathbf{a} \leq \mathbf{b}$ be vectors of functions or values and let $f \leq g$ be functions. Let $\mathbf{v}$ be a vector of functions. Then

$$eval_{\mathbf{a}}(f) \leq eval_{\mathbf{b}}(f) \qquad eval_{\mathbf{a}}(f) \leq eval_{\mathbf{a}}(g) \qquad eval_{\mathbf{a}}(eval_{\mathbf{v}}(f)) = eval_{eval_{\mathbf{a}}(\mathbf{v})}(f) \;.$$

Our contribution is a new method for solving systems of polynomial equations over $\mathcal{S}$ in several unknowns $\mathcal{X}$. A **system of equations** is a vector of polynomials $\mathbf{p}$, which we denote by $\mathbf{x} = \mathbf{p}$. A **solution** for it is a vector $\mathbf{v} \in \mathcal{S}^{\mathcal{X}}$ such that $\mathbf{v} = eval_{\mathbf{v}}(\mathbf{p})$. We always assume a vector of polynomials $\mathbf{p}$ to have components $\mathbf{p}_x$.

We will often relate languages $L \subseteq \Sigma^*$ over the alphabet $\Sigma = \mathcal{S} \cup \mathcal{X}$ of semiring elements and variables with functions. If $\mathcal{S}$ is infinite, we only need the finitely many elements that occur in the system of equations of interest. Given a word $w \in \Sigma^*$, we define $sr(w) \in \mathcal{S}[\mathcal{X}]$ to be the monomial obtained by replacing concatenation with multiplication in the semiring. Moreover, we define the function $sr(L) = \bigoplus_{w \in L} sr(w)$ summing up all monomials obtained from words in the language. In turn, given a monomial $g \in \mathcal{S}[\mathcal{X}]$, we write $wd(g) \in \Sigma^*$ for the word obtained by understanding semiring multiplication as concatenation.

3 Munchausen Iteration

We define the iteration scheme, relate it to derivation trees of context-free grammars, and with this relation derive our main theorem on convergence to the least fixed point, invoking deep results about Newton iteration from [3].

3.1 Definition

Consider a system of polynomial equations $\mathbf{x} = \mathbf{p}$. Our method works over the semiring of functions $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$. To highlight in $\mathbf{p}$ the functional aspect $\mathbf{f}$ and separate it from the constant part $\mathbf{a}$, we rewrite the system as $\mathbf{x} = \mathbf{f} + \mathbf{a}$. The first step is to compute a linear completion of the right-hand side polynomials $\mathbf{f}$. The idea is borrowed from linear algebra, namely repeatedly substituting variables y occurring in $\mathbf{f}_x$ by their defining functions $\mathbf{f}_y$. To be able to represent the resulting function in a closed form, we focus on **linear** substitutions where the next substitution is applied to a variable in $\mathbf{f}_y$.

To render this formally, a **substitution** is defined to be a pair consisting of a variable and a polynomial, denoted by $\{x \mapsto g\}$ from $\mathcal{X} \times \mathcal{S}[\mathcal{X}]$. The **application** of $\{x \mapsto g\}$ to a polynomial f yields the set of polynomials $f\{x \mapsto g\}$ containing all variants of f where one occurrence of x has been replaced by g. If x does not occur in f, $f\{x \mapsto g\}$ is empty. We are interested in substitutions that are induced by the given system of equations and that are applied repeatedly, in the aforementioned linear fashion.

▶ **Definition 1.** The set of **linear polynomial substitutions** σ_x for each variable $x \in \mathcal{X}$ associated with $\mathbf{f} \in \mathcal{S}[\mathcal{X}]^{\mathcal{X}}$ is defined mutually inductive by

$$\sigma_x ::= \{x \mapsto x\} \mid \{x \mapsto g\} \quad \text{with } g \in \mathbf{f}_x \sigma_y \ .$$

The set of **linear monomial substitutions** τ_x (for variable x) associated with $\mathbf{f}$ is defined similarly but works on monomials m_{i_x} rather than the full $\mathbf{f}_x = \bigoplus_{i_x} m_{i_x}$.

Linear polynomial substitutions yield the intuitive notion of completion we are aiming at.

▶ **Definition 2.** Consider $\mathbf{f} \in \mathcal{S}[\mathcal{X}]^{\mathcal{X}}$. Its **linear completion** $lc(\mathbf{f}) : \mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$ is defined component-wise by $lc(\mathbf{f})_x = lc(\mathbf{f}_x) = \bigoplus_{\sigma_x} x\sigma_x$. Here, σ_x ranges over all linear polynomial substitutions σ_x associated with $\mathbf{f}$.

Linear monomial substitutions do not contain sums which makes them algorithmically easier to handle than the more general linear polynomial substitutions. The following lemma shows that wrt. completion the two can be used interchangeably. The proof is by distributivity.

▶ **Lemma 3.** *Consider* $\mathbf{f} \in \mathcal{S}[\mathcal{X}]^{\mathcal{X}}$. *Then* $lc(\mathbf{f}_x) = \bigoplus_{\tau_x} x\tau_x$.

Munchausen iteration starts with the linear completion. The iteration step then inserts the current approximant into itself. Note that the method is applied only to the functional part $\mathbf{f}$ of the system of equations, and the result is a sequence of functions $\beta^{(n)}$. To obtain a value in $\mathcal{S}$, we have to evaluate $\beta^{(n)}$ at some vector $\mathbf{b} \in \mathcal{S}^{\mathcal{X}}$. An obvious choice for $\mathbf{b}$ is the given vector of constants $\mathbf{a}$. We will show that we can evaluate at any vector that lies between $\mathbf{a}$ and the least fixed point, and still converge to the least fixed point (Theorem 9).

▶ **Definition 4.** Consider $\mathbf{f} \in \mathcal{S}[\mathcal{X}]^{\mathcal{X}}$. The Munchausen iteration is

$$\beta^{(0)} \;=\; lc(\mathbf{f}) \qquad \beta^{(n+1)} \;=\; eval_{\beta^{(n)}}(\beta^{(n)}) \ .$$

The **Munchausen sequence wrt.** $\mathbf{b} \in \mathcal{S}^{\mathcal{X}}$ is defined by $\beta(\mathbf{b})^{(n)} \;=\; eval_{\mathbf{b}}(\beta^{(n)})$.

We aim to prove that the n^{th} element of the Munchausen sequence wrt. $\mathbf{a}$ is equal to the $2^{n\text{th}}$ Newton approximant. We make the link to Newton iteration using derivation trees.

3.2 Derivation Tree Analysis

Every system of polynomial equations relates naturally to a context-free grammar. We show that the n^{th} Munchausen approximant coincides with the yields of the derivation trees of dimension at most 2^n . For the development, consider the functional part $\mathbf{f} \in \mathcal{S}[\mathcal{X}]^{\mathcal{X}}$ of the system of equations of interest. We associate with it the context-free grammar $G_{\mathbf{f}} = (\mathcal{X}, S, \bigcup_{x\in\mathcal{X}} P_x)$. The variables form the non-terminals, the semiring elements give the terminal symbols. There is a set of production rules P_x for every non-terminal x. For the definition, assume $\mathbf{f}_x = \bigoplus_{i=1}^{k_x} m_i$. The productions are

$$P_x = \{x \to wd(m_i) \mid i = 1, \ldots, k_x\} \ .$$

Since $G_{\mathbf{f}}$ is a context-free grammar, we can make use of the concept of **derivation trees** (that may have variables at the leaves). Let $\mathcal{T}(x)$ denote the set of all derivation trees that can be generated from the non-terminal x. We write $\mathcal{T}_n(x)$ for the set of derivation trees in $\mathcal{T}(x)$ of dimension at most $n \in \mathbb{N}$. The **dimension** $dim(t)$ of a tree t is a well-known concept [4] and defined inductively as follows. (i) If t has no children, then $dim(t) = 0$. (ii) If t has precisely one child t_1, then $dim(t) = dim(t_1)$. (iii) If t has at least two children, consider the children t_1 and t_2 of highest dimension. More formally, let $dim(t_1) \geq dim(t_2)$ and $dim(t_2) \geq dim(t')$ for all children $t' \neq t_1$. In this case, we set

$$dim(t) \;=\; \begin{cases} dim(t_1) + 1 & \text{if } dim(t_1) = dim(t_2) \ , \\ dim(t_1) & \text{if } dim(t_1) > dim(t_2) \ . \end{cases}$$

The correspondence with derivation trees relies on the following lemma: A tree of dimension $2m$ can be decomposed into trees of dimension at most m. For the formal statement, if t' is a tree with leaves $l_1, \ldots, l_n$ (from left to right) and $t_1, \ldots, t_n$ are trees, we denote the tree obtained by replacing each leaf l_i with t_i by $t'[t_1, \ldots, t_n]$.

▶ **Lemma 5.** *Consider a tree t with $dim(t) = 2m$. Then there are trees $t', t_1, \ldots, t_n$ with $dim(t'), dim(t_1), \ldots, dim(t_n) \leq m$ so that $t = t'[t_1, \ldots, t_n]$.*

Proof. We identify the maximal subtrees t_1 to t_n of t that satisfy $dim(t_i) \leq m$. Note that they are unique. Removing them from t leaves us with a subtree t'. Tree t' has the same root as t. The leaves are labeled by $root(t_1)$ to $root(t_n)$. If we replace each leaf $root(t_i)$ by the tree t_i, we obtain a representation of t:

$$t \;=\; t'[t_1, \ldots, t_n] \ .$$

To establish $\mathit{dim}(t') \leq m$, assume towards a contradiction that $\mathit{dim}(t') > m$. Consider the children t_i that we removed from t to obtain t'. By the maximality requirement for t_i, the parent node of $\mathit{root}(t_i)$ in t has outdegree ≥ 2.

Assume for every child t_i one of the following holds: (i) $\mathit{dim}(t_i) = m$ or (ii) t_i has a sibling of dimension $> m$ or (iii) t_i has two siblings of dimension $= m$. In each of the cases, we can assume t_i to contribute a dimension of m when we determine $t'[t_1, \ldots, t_n]$. As a result, we obtain

$$\mathit{dim}(t) = \mathit{dim}(t') + m > m + m \ .$$

This contradicts the assumption that the dimension of t equals $2m$.

As a consequence of this contradiction, there has to be a child t_i that does not satisfy any of (i) to (iii) above. This means $\mathit{dim}(t_i) < m$, there is no sibling of dimension $> m$, and there is at most one sibling of dimension m. Let x be the parent node of $\mathit{root}(t_i)$. Let t_x be the subtree with x as its root. The violation of (i) to (iii) allows us to conclude $\mathit{dim}(t_x) \leq m$. This in turn contradicts the maximality of t_i.

Since we derived a contradiction in both cases (all children satisfy (i), (ii), or (iii) and there is a child that does not satisfy (i), (ii), and (iii)), we have to conclude that the assumption $\mathit{dim}(t') > m$ has to be false. ◀

The correspondence is our first main result. Recall that the yield of a given tree, $\mathit{yield}(t)$, is the word formed by the leaves when the tree is traversed in left-first manner.

▶ **Theorem 6.** $\beta^{(n)}{}_x \ = \ \bigoplus_{t \in \mathcal{T}_{2^n}(x)} \mathit{sr}(\mathit{yield}(t))$.

Proof. We proceed by induction on n.

Base case $n = 0$ In the base case, we have

$$\beta^{(0)}{}_x = \mathit{lc}(\mathbf{f}_x) = \bigoplus_{\sigma_x} x\sigma_x = \bigoplus_{\tau_x} x\tau_x \ .$$

The first two equalities are by definition, the last is Lemma 3. We have to show that

$$\bigoplus_{\tau_x} x\tau_x = \bigoplus_{t \in \mathcal{T}_1(x)} \mathit{sr}(\mathit{yield}(t)) \ .$$

To see that each $x\tau_x$ can be obtained as the yield of a derivation tree rooted in x, note that the substitution τ_x takes the form

$$\{x \mapsto m_x\{\ldots\{z \mapsto m_z\}\}\ldots\} \ .$$

Here, m_x to m_z are monomials and so $x \to m_x$ up to $z \to m_z$ are production rules in the grammar $G_{\mathbf{f}}$. Hence, the application of τ_x corresponds to a derivation sequence from x.

We show that the derivation tree has dimension at most one. Since the substitution is linear, we can highlight in every rule the variable that will be replaced next. So in the initial step, we have $x \to \mathit{wd}(m_i)$ with $m_i = g_1 \odot y \odot g_2$. In the derivation tree, the rule yields several subtrees for root x. Since g_1 and g_2 are monomials, their subtrees consist of single nodes labeled by an element from the semiring or a variable. The remaining subtree is for y and the same reasoning applies. Since the subtrees labeled by a semiring element and the subtrees of a variable have dimension zero, the overall derivation tree has dimension at most one (zero, if no rule is applied).

For the reverse direction, we have to show that for every subtree $t \in \mathcal{T}(x)$ of dimension at most one, we have a linear monomial substitution τ_x that we can apply to x to obtain

$sr(yield(t))$. The first observation is that in t, for every pair of siblings at least one has to have dimension zero. Assume this was not the case and both siblings have dimension at least one. In this case, their parent node has dimension at least two which contradicts the assumption on the dimension of t. The only trees with dimension zero are linear paths. Since there are no productions of the shape $x \to y$ in $G_{\mathbf{f}}$ (since we removed the constants from $\mathbf{f}$ and all other monomials are at least of the shape $a_1 \odot y \odot a_2$) the trees of dimension zero have to be leaves. Combined with the fact that one sibling has to have dimension zero, we obtain that t is a path with semiring elements and variables to the sides. Hence, it forms a linear monomial substitution. Note that this covers the case where the path has dimension zero. Then the tree is x itself, to which we apply $\{x \mapsto x\}$.

Induction step Assume ${\beta^{(n)}}_x = \bigoplus_{t \in \mathcal{T}_{2^n}(x)} sr(yield(t))$ holds and consider $n+1$.

The following equations make use of the definition of $\beta^{(n+1)}$, the induction hypothesis, and the definition of evaluation:

$$\begin{aligned} {\beta^{(n+1)}}_x &= eval_{\beta^{(n)}}({\beta^{(n)}}_x) \\ &= eval_{\beta^{(n)}}(\bigoplus_{t \in \mathcal{T}_{2^n}(x)} sr(yield(t))) \\ &= \bigoplus_{t \in \mathcal{T}_{2^n}(x)} eval_{\beta^{(n)}}(sr(yield(t))) . \end{aligned}$$

The evaluation $eval_{\beta^{(n)}}(sr(yield(t)))$ replaces every variable y in $sr(yield(t))$ by ${\beta^{(n)}}_y$. By the induction hypothesis, ${\beta^{(n)}}_y = \bigoplus_{t \in \mathcal{T}_{2^n}(y)} sr(yield(t))$. This means every variable y is replaced by the sum of the yields of all derivation trees t' with $dim(t') \leq 2^n$. By ω-continuity, we can equivalently sum up all monomials that result from $sr(yield(t))$ by replacing y by the yield of a single derivation tree t'.

To establish the inequality ${\beta^{(n+1)}}_x \leq \bigoplus_{t \in \mathcal{T}_{2^{n+1}}(x)} sr(yield(t))$, note that every monomial of $eval_{\beta^{(n)}}(sr(yield(t)))$ is obtained from a derivation tree t'' which equals t but appends the trees t' to the leaves. Since t as well as the t' have dimension at most 2^n, the resulting tree t'' has dimension at most $2^n + 2^n = 2^{n+1}$.

For the reverse direction, we show $sr(yield(t)) \leq {\beta^{(n+1)}}_x$. Consider a derivation tree $t \in \mathcal{T}(x)$ of dimension 2^{n+1}. The same argumentation holds for trees of smaller dimension. By Lemma 5, the tree can be decomposed into t' and $t_1, \ldots t_n$, all of dimension at most 2^n:

$$t \;=\; t'[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] \;.$$

By definition of the yield, we get that $yield(t)$ results from $yield(t')$ by replacing x_1 to x_n with $yield(t_1)$ to $yield(t_n)$, respectively. The above discussion concludes the case. ◀

3.3 Results

We prove that the Munchausen sequence converges to the least fixed point. Moreover, in the commutative case it is guaranteed to reach the least fixed point in a number of steps that is logarithmic in the number of variables. Both results rely on a precise correspondence between Munchausen iteration and Newton iteration.

To define the Newton iteration, we recall the concept of **differentials**. The differential of a polynomial p wrt. a variable $x \in \mathcal{X}$ at point $\mathbf{v}$ is the polynomial defined inductively by

$$D_x p|_{\mathbf{v}} = \begin{cases} \bigoplus_{i \in I} D_x m_i|_{\mathbf{v}} & \text{if } p = \bigoplus_{i \in I} m_i \ , \\ (D_x g|_{\mathbf{v}} \odot eval_{\mathbf{v}}(h)) \oplus (eval_{\mathbf{v}}(g) \odot D_x h|_{\mathbf{v}}) & \text{if } p = g \odot h \ , \\ 0 & \text{if } p \in S \text{ or } p \in \mathcal{X} \setminus \{x\} \ , \\ x & \text{if } p = x \ . \end{cases}$$

The differential of p at point $\mathbf{v}$ is the sum $Dp|_{\mathbf{v}} = \bigoplus_{x \in \mathcal{X}} D_x p|_{\mathbf{v}}$. The differential of a vector of polynomials is defined component-wise, $(D\mathbf{p}|_{\mathbf{v}})_x = D\mathbf{p}_x|_{\mathbf{v}}$. The function $D\mathbf{f}|_{\mathbf{v}}{}^*$ is defined by summing up all i-fold applications of the differential, i.e. $D\mathbf{f}|_{\mathbf{v}}{}^* = \bigoplus_{i \in \mathbb{N}} D\mathbf{f}|_{\mathbf{v}}{}^i$ with $D\mathbf{f}|_{\mathbf{v}}{}^0 = id$ and $D\mathbf{f}|_{\mathbf{v}}{}^{i+1} = eval_{D\mathbf{f}|_{\mathbf{v}}{}^i}(D\mathbf{f}|_{\mathbf{v}})$.

With differentials at hand, the **Newton iteration** is

$$\nu^{(0)} \ = \ eval_{\mathbf{0}}(\mathbf{p}) \qquad \nu^{(n+1)} \ = \ eval_{\nu^{(n)}}(D\mathbf{p}|_{\nu^{(n)}}{}^*) \ .$$

Actually, this is not the most general definition of Newton iteration but coincides with it in the idempotent case that we consider. An explanation of why the sequence mimics the classical method from numerics is beyond the scope of this paper. It can be found in [3].

The n^{th}-Newton approximant is known to correspond to the derivation trees of dimension at most n. To be precise, Esparza et al. consider **complete** derivation trees where the yields do not contain variables.[1] Let $G_{\mathbf{f}}(\mathbf{a})$ be the grammar that adds to $G_{\mathbf{f}}$ the rules $x \to \mathbf{a}_x$ for each variable. Let $\mathcal{C}_n(x)$ denote the set of complete derivation trees of dimension at most n from non-terminal x in $G_{\mathbf{f}}(\mathbf{a})$.

▶ **Theorem 7** (Esparza et al. [3]). $\nu^{(n)}{}_x \ = \ \bigoplus_{t \in \mathcal{C}_n(x)} sr(yield(t))$.

We argue that the complete trees of dimension n of $G_{\mathbf{f}}(\mathbf{a})$ are precisely the (incomplete) trees of dimension n of $G_{\mathbf{f}}$, extended by appending the constants. Appending the constants means to every leaf labeled by x we append a child node $\mathbf{a}_x$. To see the correspondence, note that removing or adding those appendices does not change the dimension. The semiring element corresponding to the yield of the extended tree is precisely the semiring element for the yield of the original tree evaluated at the vector $\mathbf{a}$.

▶ **Lemma 8.** $eval_{\mathbf{a}}(\bigoplus_{t \in \mathcal{T}_{2^n}(x)} sr(yield(t))) \ = \ \bigoplus_{t \in \mathcal{C}_{2^n}(x)} sr(yield(t))$.

We can now show that the n^{th} element of the Munchausen sequence wrt. $\mathbf{a}$ equals the $2^{n\text{th}}$ Newton approximant. Since the Newton sequence converges to the least fixed point $\mu\mathbf{p}$, so does the Munchausen sequence. Evaluating at larger vectors $\mathbf{b}$ requires further arguments.

▶ **Theorem 9.** *Let* $\mathbf{x} = \mathbf{p} = \mathbf{f} + \mathbf{a}$ *be a system of polynomial equations.*

(1) $\beta(\mathbf{a})^{(n)} = \nu^{(2^n)}$.
(2) Let $\mathbf{a} \leq \mathbf{b} \leq \mu\mathbf{p}$. *Then* $\sup_{n \in \mathbb{N}} \beta(\mathbf{b})^{(n)} = \mu\mathbf{p}$.

Proof. We show (1). Using Theorem 6, Lemma 8, and Theorem 7 yields

$$\beta(\mathbf{a})^{(n)}{}_x = eval_{\mathbf{a}}(\beta^{(n)}{}_x) = eval_{\mathbf{a}}\big(\bigoplus_{t \in \mathcal{T}_{2^n}(x)} sr(yield(t))\big) = \bigoplus_{t \in \mathcal{C}_{2^n}(x)} sr(yield(t)) = \nu^{(2^n)}{}_x \ .$$

◀

[1] To handle the non-idempotent case, the trees are also decorated. We elaborate on this in Section C.

In the commutative case, we can apply another deep result from [3]: The number of iterations needed to reach the least fixed is at most the number of variables in $\mathcal{X}$.

▶ **Corollary 10.** *If $\mathcal{S}$ is commutative, we have $\mu\mathbf{p} = \beta(\mathbf{a})^{(\lceil \log |\mathcal{X}| \rceil)}$.*

3.4 Related Methods

We already elaborated on the relationship with Newton iteration and with Kleene iteration. An improvement of Newton iteration to a hierarchy (in terms of convergence speed) of iteration schemes appeared in [2]. The idea is to repeatedly apply the Newton operator to itself. The main result shows that one application of the n-fold Newton operator and n steps of Newton iteration coincide.

The hierarchy of Newton iterations is substantially different from the Munchausen iteration we present here. It relies on a linear derivation process that adds one dimension with each self application. In this (outer) derivation a result of dimension n is inserted, leading to a result of dimension $n+1$. Munchausen iteration inserts a derivation result of dimension n into a derivation of dimension n, thus doubling the analysis information in every step.

4 Algorithmic Considerations

We study the operations of linear completion and evaluation as they are needed for the initial and for the iteration step of the Munchausen scheme.

4.1 Linear Completion

As indicated by the correspondence between the 0th Munchausen approximant and the 1st Newton approximant, the differential $D\mathbf{f}$ should be a possibility to represent the linear completion of $\mathbf{f}$. To be precise, we need to sum up all i-fold applications of the differential to obtain the linear completion. The proof shows that the i-fold application corresponds to all monomial substitutions of length $i+1$.

▶ **Theorem 11.** *For every vector $\mathbf{v} \in \mathcal{S}^{\mathcal{X}}$, we have $eval_{\mathbf{v}}(lc(\mathbf{f})) = eval_{\mathbf{v}}(D\mathbf{f}|_{\mathbf{v}}{}^{*})$.*

We now show how to construct a linear context-free grammar that represents the linear completion. The benefit over Theorem 11 is that we are not bound to using differentials but have available the spectrum of language-theoretic techniques — even for regular languages (Section 4.3). By Lemma 3, the linear completion is (for each variable) the sum

$$lc(\mathbf{f}_x) = \bigoplus_{\tau_x} x\tau_x \qquad \text{where } \tau_x \text{ has the form} \qquad \tau_x = \{x \mapsto m_{i_x}\{y \mapsto m_{i_y}\{\ldots\}\}\} \; .$$

By the definition of linear substitutions, after $x \mapsto m_{i_x}$ the next substitution $y \mapsto m_{i_y}$ will be applied to a single occurrence of y in m_{i_x}. The idea of the grammar construction is to highlight in each monomial the variable that will be replaced next. To be precise, we even fix the occurrence of the variable that will be rewritten. Given a monomial m and an occurrence z of a variable in m, there are unique monomials $m^{z,l}$ and $m^{z,r}$ so that

$$m = m^{z,l} \odot z \odot m^{z,r} \; . \tag{1}$$

We define the grammar to be $LG^{(0)} = (\{y^{(1)} \mid y \in \mathcal{X}\}, \mathcal{X} \cup \mathcal{S}, \bigcup_{y\in\mathcal{X}} P_y \cup P)$. We create a non-terminal (with index) for each variable. The terminals are the variables and the semiring elements. The reason $LG^{(0)}$ has non-terminals $y^{(1)}$ is that we will see an exponential growth in the number of non-terminals during evaluation when we make the grammars explicit (Section 4.2). Every monomial m of $\mathbf{f}_y$ and every occurrence z of a variable in m will induce a rule that mimics the decomposition in Equation (1). Note that all variables in $m^{z,l}$ and $m^{z,r}$ are terminals, which reflects the fact that they will not be replaced by further linear substitutions. Moreover, note that a variable may have several occurrences in m, in which case we obtain several rules:

$$P_y = \{y^{(1)} \to wd({m_{i_y}}^{z,l}) \cdot z^{(1)} \cdot wd({m_{i_y}}^{z,r}) \mid \mathbf{f}_y = \bigoplus_{i_y} m_{i_y}, z \text{ an occurrence in } m_{i_y}\} \ .$$

The productions $P = \{y^{(1)} \to y \mid y \in \mathcal{X}\}$ mimic the identity substitution. We obtain a one-to-one correspondence between the linear substitutions applied to x and the sentential forms derivable from $x^{(1)}$, denoted by $L(LG_x^{(0)})$.

▶ **Proposition 12.** $lc(\mathbf{f}_x) = sr(L(LG_x^{(0)}))$.

Computing information from $L(LG_x^{(0)})$ is still non-trivial since we do not have a closed expression for the language. There are two special cases when $L(LG_x^{(0)})$ is easy to evaluate. If $\mathcal{S}$ is finite, also the set of functions $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}$ is finite. In this setting, a Kleene iteration applied to $LG_x^{(0)}$ (more precisely, a system of linear equations obtained from the grammar) is sufficient to determine a closed-form description of the linear completion.

If $\mathcal{S}$ is commutative, the grammar construction can be modified to ensure left-linearity. Indeed, Equation (1) simplifies to the following unique representation of a monomial m wrt. a variable z (we no longer have to work with variable occurrences):

$$m = z \odot m^z \ . \tag{2}$$

This in turn simplifies the transitions to $y^{(1)} \to z^{(1)} \cdot wd(m_{i_y}^z)$.

The left-linear grammar yields a closed representation of the linear completion as a regular expression over $\mathcal{X} \cup \mathcal{S}$, on which further evaluation steps can be performed. Actually, we only need the Parikh image of the language [12], which is a semilinear set and potentially more compact.

4.2 Evaluation

To capture $\beta^{(n+1)}$, we show how to reflect $eval_{\beta^{(n)}}(\beta^{(n)})$ on grammar level. Assume we have a grammar $LG^{(n)}$ with language $\beta^{(n)}$. Our construction will maintain the invariant that $LG^{(n)}$ has non-terminals of the form $y^{(m)}$ with $1 \leq m \leq 2^n$. The terminals will always be $\mathcal{X} \cup \mathcal{S}$. The grammar for $\beta^{(n+1)}$ will behave like $LG^{(n)}$ but invoke itself when it reaches a terminal y. To invoke y, we have to turn the variable into a non-terminal. We create two copies of $LG^{(n)}$ and modify the indices in one of the copies. This index shift in particular turns a former terminal y into $y^{(2^n)}$, which is a non-terminal in the other grammar:

$$LG^{(n+1)} = LG^{(n)} \cup (LG^{(n)} + 2^n) \ .$$

Formally, the **index shift by** $k \in \mathbb{N}$ turns $LG^{(n)}$ into the grammar $LG^{(n)} + k$, where consistently all non-terminal indices are increased by k and all terminals y are turned into non-terminals $y^{(k)}$. To give an example, the production $y^{(i)} \to a \cdot x \cdot z^{(i)}$ from $LG^{(n)}$ will be turned into $y^{(i+k)} \to a \cdot x^{(k)} \cdot z^{(i+k)}$ in $LG^{(n)} + k$. The union of the grammars is taken componentwise. Let the sentential forms derivable from $x^{(2^n)}$ be denoted by $L(LG_x^{(n)})$.

▶ **Proposition 13.** *For each $n \in \mathbb{N}$, we have* ${\beta^{(n)}}_x = sr(L(LG_x^{(n)}))$.

To get from the Munchausen iteration to the Munchausen sequence, we need to evaluate the function $\beta^{(n)}$ at a vector of constants $\mathbf{b}$. This operation can also be performed on the grammar. We treat the occurrences of y as non-terminals instead of terminals and add the rules $y \to \mathbf{b}_y$ for every variable $y \in \mathcal{X}$. Let the resulting grammar be $LG(\mathbf{b})^{(n)}$.

▶ **Proposition 14.** *For each $n \in \mathbb{N}$, we have $\beta(\mathbf{b})^{(n)}{}_x \;=\; sr(L(LG(\mathbf{b})_x^{(n)}))$.*

The grammars $LG^{(n)}$ have productions of the same shape that only differ in the index n. We exploit this to give a more compact representation of the language by an **indexed grammar**. Indexed grammars annotate the non-terminals in the productions with a stack.

The indexed grammars IG we define uses the same non-terminals and terminals as $LG^{(0)}$. The stack $s \in 1^*0$ encodes the index in unary. The set of production rules is $\bigcup_{y\in\mathcal{X}} R_y \cup R$. As in P_y, the productions in R_y start in $y^{(1)}$ and single out one occurrence $z^{(1)}$ of a variable in a monomial of $\mathbf{f}_y$. When using the rule, the stack $[1.s]$ of $y^{(1)}$ is passed to $z^{(1)}$. Also the other variables are treated as non-terminals. For them, the stack height is decreased by one. Formally, for each occurrence z of a variable in a monomial m_{i_y} of $\mathbf{f}_y$, the set R_y has a rule

$$y^{(1)}[1.s] \to wd({m_{i_y}}^{z,l})[s] \;\cdot\; z^{(1)}[1.s] \;\cdot\; wd({m_{i_y}}^{z,r})[s] \;.$$

The set R contains a rule for each variable that replaces the non-terminal version by the terminal version if the stack is empty, $R = \{y^{(1)}[0] \to y \mid y \in \mathcal{X}\}$.

We define $L(IG_x^{(n)})$ to be the set of sentential forms derivable in IG from $x^{(1)}[2^n]$, i.e. with the unary encoding of 2^n as initial stack content. Obviously, $L(IG_x^{(n)}) = L(LG_x^{(n)})$, and we can also perform the evaluation by adding rules as for $LG^{(n)}$. This allows us to phrase the Propositions 13 and 14 in terms of the indexed grammar IG.

4.3 Tensor Semirings

Left-linear grammars are preferrable over linear context-free ones for the better algorithmics they support (see below). We show that we can work with left-linear grammars also in the case of non-commutative io-semirings. To this end, we adapt the recent work [15]. Reps et al. have shown that — provided the semiring of interest has an associated tensor-product semiring — every system of linear equations over the semiring can be transformed to a left-linear system over the tensor-product semiring. One important example where a tensor-product semiring exists is predicate abstraction [15].

▶ **Definition 15.** We call an io-semiring $\mathcal{S}$ **admissible**, if there is a transpose operation, an associated tensor-product semiring, and a readout operation.

The **transpose** $\cdot^t : \mathcal{S} \to \mathcal{S}$ should satisfy

$$(a \oplus b)^t = a^t \oplus b^t \qquad (a \odot b)^t = b^t \odot a^t \qquad (a^t)^t = a \;.$$

A **tensor-product semiring** $\mathcal{S}_\mathcal{T}$ is an io-semiring $(\mathcal{S}_\mathcal{T}, \oplus_\mathcal{T}, \odot_\mathcal{T}, 0_\mathcal{T}, 1_\mathcal{T})$ together with a map $\otimes : \mathcal{S} \times \mathcal{S} \to \mathcal{S}_\mathcal{T}$ such that

$$0 \otimes a = a \otimes 0 = 0_\mathcal{T} \qquad (a \otimes b) \odot_\mathcal{T} (c \otimes d) = (a \odot c) \otimes (b \odot d)$$
$$a \otimes (b \oplus c) = (a \otimes b) \oplus_\mathcal{T} (a \otimes c) \qquad (b \oplus c) \otimes a = (b \otimes a) \oplus_\mathcal{T} (c \otimes a) \;.$$

The **readout** operation $\mathcal{R} : \mathcal{S}_\mathcal{T} \to \mathcal{S}$ should satisfy (with I finite or countable)

$$\mathcal{R}(a \otimes b) = a^t \odot b \qquad \mathcal{R}(\bigoplus_{i\in I} p_i) = \bigoplus_{i\in I} \mathcal{R}(p_i) \;.$$

The crucial requirement is the existence of a readout operation that distributes over sums without producing cross terms. It is, for example, not met by the language semiring.

Consider a system of linear equations over an admissible semiring $\mathcal{S}$ of the form

$$x_i = c_i \oplus \bigoplus_{j=1,\dots,k} a_{i,j} \odot x_j \odot b_{i,j} \quad \text{for } i = 1, \dots, k \ . \tag{3}$$

Reps et al. define its **regularization** to be the left-linear system over the associated tensor-product semiring $\mathcal{S}_{\mathcal{T}}$:

$$y_i = (1^t \otimes c_i) \oplus_{\mathcal{T}} \bigoplus\nolimits_{\mathcal{T}}{}_{j=1,\dots,k} y_j \odot_{\mathcal{T}} (a^t_{i,j} \otimes b_{i,j}) \quad \text{for } i = 1, \dots, k \ . \tag{4}$$

Their main result shows that the least solution to (3) can be obtained from the least solution to (4) by applying the readout operation.

▶ **Theorem 16** (Reps et al. [15]). *Let* $\mathbf{v}$ *be the least solution to (4). Then* $\mathcal{R}(\mathbf{v})$ *is the least solution to (3).*

The importance of the result stems from the fact that systems of left-linear equations (4) enjoy efficient algorithmics. For example, Tarjan's path-expression algorithm [18] can be applied to (4) to obtain for every y_i a regular expression (over the tensor-product semiring) capturing the least solution. We discuss how to use this in our setting.

Consider the system of linear equations for the linear completion that is obtained from $LG^{(0)}$. Let $\theta^{(0)}$ denote its regularization. With Tarjan's algorithm, we obtain for $\theta^{(0)}$ a regular expressions over the tensor semiring and $\mathcal{X}$. As a consequence of Theorem 16 and Proposition 12, we have $lc(\mathbf{f}) = \mathcal{R}(\theta^{(0)})$.

One would also like to carry out the evaluation process over the tensor-product semiring. Unfortunately, $\theta^{(0)}$ is a regular expression with variables denoting elements from $\mathcal{S}$, namely those occurrences that were treated as terminals by the grammar. Therefore, we cannot evaluate $\theta^{(0)}$ at $\theta^{(0)}$, but only at $\mathcal{R}(\theta^{(0)})$. We define $\theta^{(n+1)} = eval_{\mathcal{R}(\theta^{(n)})}(\theta^{(n)})$. Using Theorem 16 and induction, we get $\beta^{(n)} = \mathcal{R}(\theta^{(n)})$ for all $n \in \mathbb{N}$. For an implementation, the idea would be to nevertheless insert the tensor element $\theta^{(0)}$ and define a recursive readout.

5 Discussion

We gave a new iteration scheme for solving polynomial equations over ω-continuous and idempotent semirings. The key idea is to solve the equations over the semiring of functions rather than the semiring of interest and only evaluate the resulting function when needed. We showed that the method is exponentially faster than the well-known Newton sequence [3], and that we can obtain symbolic descriptions for the solutions. The descriptions can be understood as identifying maximal sharing in the derivation trees of context-free grammars.

Unfortunately, we do not yet know how to handle these descriptions. If we give them explicitly as linear context-free grammars, semilinear sets, or regular expressions over the tensor semiring, the size of the description doubles in every step. Hence, we buy an exponential improvement in time at the cost of an exponential blow up in space. This still means we compute a description of size n in $log\ n$ steps. Experiments will have to tell how this compares to Newton iteration that, for the same result, needs n steps but where the objects are semiring values rather than grammars.

The descriptions we obtain are structured to an extent that allows us to represent them symbolically, by a restricted class of indexed grammars (over linear context-free grammars, semilinear sets, or regular expressions). With restricted indexed grammars, the iteration steps of Munchausen are easy to compute. The drawback is that, so far, we do not know how to extract information from the restricted indexed grammars. As future work, we plan to understand how to compute in such highly symbolic structures.

Acknowledgments

We thank Stefan Kiefer for helpful discussions.

References

1 P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Artificial Intelligence and Programming Languages*, pages 1–12. ACM, 1977.

2 J. Esparza, S. Kiefer, and M. Luttenberger. On fixed point equations over commutative semirings. In *STACS*, volume 4393 of *LNCS*, pages 296–307. Springer, 2007.

3 J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian program analysis. *JACM*, 57(6), 2010.

4 J. Esparza, M. Luttenberger, and M. Schlund. A brief history of Strahler numbers. In *LATA*, volume 8370 of *LNCS*, pages 1–13. Springer, 2014.

5 J. Esparza, M. Luttenberger, and M. Schlund. FPSOLVE: A generic solver for fixpoint equations over semirings. In *CIAA*, volume 8587 of *LNCS*, pages 1–15. Springer, 2014.

6 K. Etessami and M. Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *JACM*, 56(1), 2009.

7 M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, pages 471–482. ACM, 2010.

8 L. Holik and R. Meyer. Antichains for the verification of recursive programs. In *NETYS*, volume 9466 of *LNCS*, pages 322–336. Springer, 2015.

9 M. W. Hopkins and D. Kozen. Parikh's theorem in commutative Kleene algebra. In *LICS*, pages 394–401. IEEE, 1999.

10 U. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

11 F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

12 R. Parikh. On context-free languages. *JACM*, 13(4), 1966.

13 S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

14 T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM, 1995.

15 T. Reps, E. Turetsky, and P. Prabhu. Newtonian program analysis via tensor product. In *POPL*, pages 663–677. ACM, 2016.

16 H. Seidl, R. Wilhelm, and S. Hack. *Compiler Design: Analysis and Transformation.* Springer, 2012.

17 M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. Technical Report 2, New York University, 1978.

18 R. E. Tarjan. Fast algorithms for solving path problems. *JACM*, 28(3), 1981.

A Proofs of Section 3

Proof of Lemma 3. Inequality $\geq$ is immediate by the fact that for every linear monomial substitution, there is a linear polynomial substitution that always inserts the full polynomial rather than one of its monomials. To show $\leq$, assume that every substitution in σ_x applies to precisely one position so that $f\sigma_x$ is a single function and not a set. The general case follows since we do not make an assumption of where the application occurs. Substitution σ_x has the shape

$$\{x \mapsto \mathbf{f}_x\{y \mapsto \mathbf{f}_y\{\ldots\{z \mapsto \mathbf{f}_z\}\ldots\}\}\} \; .$$

If we assume $\mathbf{f}_x = \bigoplus_{i_x} m_{i_x}$ and similarly for $\mathbf{f}_y$ and $\mathbf{f}_z$, then $x\sigma_x$ is of the form

$$\bigoplus_{i_x \neq i} m_{i_x} \oplus \Big(m_{i,1} \odot \Big(\bigoplus_{i_y \neq j} m_{i_y} \oplus \big(m_{j,1} \odot (\ldots \odot (\bigoplus_{i_z} m_{i_z}) \odot \ldots) \odot m_{j,2}\big)\Big) \odot m_{i,2}\Big) \; .$$

Distributivity yields

$$\bigoplus_{i_x \neq i} m_{i_x} \oplus \bigoplus_{i_y \neq j} (m_{i,1} \odot m_{i_y} \odot m_{i,2}) \oplus \ldots \oplus \bigoplus_{i_z} (m_{i,1} \odot m_{j,1} \odot \ldots \odot m_{i_z} \odot \ldots \odot m_{j,2} \odot m_{i,2}) \; .$$

Each of these monomials is obtained by applying a linear monomial substitution to x, ranging from the identity substitution $\{x \mapsto x\}$ (not shown) over inserting some m_{i_x} from $\mathbf{f}_x$ to a long substitution that ends with a monomial m_{i_z}. Since all these substitutions are covered by the sum over τ_x, we obtain the desired inequality. ◀

Proof of Theorem 9(2). By monotonicity of *eval* in the evaluation point, we obtain the inequality $eval_{\mathbf{b}}(\beta^{(n)}) \geq eval_{\mathbf{a}}(\beta^{(n)})$ for all $n \in \mathbb{N}$. Together with Theorem 9(1), this yields

$$\sup_{n\in\mathbb{N}} eval_{\mathbf{b}}(\beta^{(n)}) \geq \sup_{n\in\mathbb{N}} eval_{\mathbf{a}}(\beta^{(n)}) = \mu\mathbf{p} \; .$$

It remains to establish $\sup_{n\in\mathbb{N}} eval_{\mathbf{b}}(\beta^{(n)}) \leq \mu\mathbf{p}$. We show that this inequality holds for $\mathbf{b} = \mu\mathbf{p}$. The desired statement then follows by monotonicity. In fact, it is enough to show that $eval_{\mu\mathbf{p}}(\beta^{(n)}) \leq \mu\mathbf{p}$ for all $n \in \mathbb{N}$. We proceed by induction on n.

In the base case, we establish $eval_{\mu\mathbf{p}}(lc(\mathbf{f}_x)) \leq (\mu\mathbf{p})_x$ simultaneously for all variables. Using Lemma 3, it sufficient to prove $eval_{\mu\mathbf{p}}(\bigoplus_{\tau_x} x\tau_x) \leq (\mu\mathbf{p})_x$, where τ_x ranges over all linear monomial substitutions for x.

We first show that for all x, for all monomials m_i of $\mathbf{f}_x = \bigoplus_{i_x} m_{i_x}$, and for all τ_y where y occurs in m_i, we have that $eval_{\mu\mathbf{p}}(g) \leq (\mu\mathbf{p})_x$ for every $g \in m_i\tau_y$. We proceed by an induction on the structure of τ_y. In the base case, $\tau_y = \{y \mapsto y\}$, $g = m_i$, and thus

$$\begin{aligned} eval_{\mu\mathbf{p}}(g) &= eval_{\mu\mathbf{p}}(m_i) \\ &\leq eval_{\mu\mathbf{p}}(m_i) \oplus eval_{\mu\mathbf{p}}(\bigoplus_{i_x \neq i} m_{i_x}) \oplus \mathbf{a}_x = eval_{\mu\mathbf{p}}(\mathbf{p}_x) = (\mu\mathbf{p})_x \; . \end{aligned}$$

The last equality holds as $\mu\mathbf{p}$ is a fixed point of $\mathbf{p}$. Assume $\tau_y = \{y \mapsto h\}$ with $h \in m_{i_y}\tau_z$. Let furthermore $m_i = m_1 \odot y \odot m_2$ so that y is replaced to obtain g. We have

$$\begin{aligned} eval_{\mu\mathbf{p}}(g) &= eval_{\mu\mathbf{p}}(m_1 \odot h \odot m_2) \\ &= eval_{\mu\mathbf{p}}(m_1) \odot eval_{\mu\mathbf{p}}(h) \odot eval_{\mu\mathbf{p}}(m_2) \\ &\leq eval_{\mu\mathbf{p}}(m_1) \odot (\mu\mathbf{p})_y \odot eval_{\mu\mathbf{p}}(m_2) \\ &= eval_{\mu\mathbf{p}}(m_1) \odot eval_{\mu\mathbf{p}}(y) \odot eval_{\mu\mathbf{p}}(m_2) \\ &= eval_{\mu\mathbf{p}}(m_1 \odot y \odot m_2) \\ &= eval_{\mu\mathbf{p}}(m_i) \leq (\mu\mathbf{p})_x \; . \end{aligned}$$

The first inequality is by the induction hypothesis combined with monotonicity, the second inequality is proven in the base case.

We can now derive $eval_{\mu\mathbf{p}}(\bigoplus_{\tau_x} x\tau_x) \leq (\mu\mathbf{p})_x$ by showing $eval_{\mu\mathbf{p}}(x\tau_x) \leq (\mu\mathbf{p})_x$ for all τ_x and by using idempotence. If $\tau_x = \{x \mapsto x\}$, we have $eval_{\mu\mathbf{p}}(x\tau_x) = eval_{\mu\mathbf{p}}(x) = (\mu\mathbf{p})_x$. If the substitution is $\tau_x = \{x \mapsto g\}$ with $g \in m_{i_x}\tau_y$, we use the statement proven above to conclude $eval_{\mu\mathbf{p}}(x\tau_x) = eval_{\mu\mathbf{p}}(g) \leq (\mu\mathbf{p})_x$.

Let us now assume that the statement holds for n. By definition and associativity, we get $eval_{\mu\mathbf{p}}(\beta^{(n+1)}) = eval_{\mu\mathbf{p}}(eval_{\beta^{(n)}}(\beta^{(n)})) = eval_{eval_{\mu\mathbf{p}}(\beta^{(n)})}(\beta^{(n)})$. Using the induction hypothesis together with monotonicity, this is at most $eval_{\mu\mathbf{p}}(\beta^{(n)})$. Applying the induction hypothesis again yields the desired inequality. ◀

B Proofs of Section 4

Proof of Theorem 11. We establish $eval_{\mathbf{v}}(lc(\mathbf{f}_x)) = eval_{\mathbf{v}}((D\mathbf{f}|_{\mathbf{v}}{}^*)_x)$ simultaneously for all components. Using Lemma 3, we have $lc(\mathbf{f}_x) = \bigoplus_{\tau_x} x\tau_x$, where τ_x ranges over all linear monomial substitutions for x. Recall the definitions

$$D_x p|_{\mathbf{v}} = \begin{cases} \bigoplus_{i\in I} D_x m_i|_{\mathbf{v}} & \text{if } p = \bigoplus_{i\in I} m_i \ , & (5)\\ (D_x g|_{\mathbf{v}} \odot eval_{\mathbf{v}}(h)) \oplus (eval_{\mathbf{v}}(g) \odot D_x h|_{\mathbf{v}}) & \text{if } p = g \odot h \ , & (6)\\ 0 & \text{if } p \in S \text{ or } p \in \mathcal{X} \setminus \{x\} \ , & (7)\\ x & \text{if } p = x \ . & (8) \end{cases}$$

and

$$D\mathbf{f}|_{\mathbf{v}}{}^* = \bigoplus_{i\in\mathbb{N}} D\mathbf{f}|_{\mathbf{v}}{}^i, \quad \text{where} \quad D\mathbf{f}|_{\mathbf{v}}{}^0 = id, \quad D\mathbf{f}|_{\mathbf{v}}{}^{i+1} = eval_{D\mathbf{f}|_{\mathbf{v}}{}^i}(D\mathbf{f}|_{\mathbf{v}}) \ .$$

We start by proving $\leq$. First note that for $\tau_x = \{x \mapsto x\}$ with $x\tau_x = x$ we also have the summand $(D\mathbf{f}|_{\mathbf{v}}{}^0)_x = id_x = x$ in $D\mathbf{f}|_{\mathbf{v}}{}^*$. In general, by summand we mean a part of the sum that forms the differential. To complete this part of the proof, we show by induction that for every $m_x\tau_y$, there is an i and a summand s of $D\mathbf{f}_x|_{\mathbf{v}}{}^i$ such that they evaluate to the same result under $\mathbf{v}$. Let us write $m_x = m_1 \odot y \odot m_2$, where y is the occurrence that will be replaced by τ_y.

In the base case, let $\tau_y = \{y \mapsto y\}$ and thus $m_x\tau_y = m_x$. Recall that $(D\mathbf{f}|_{\mathbf{v}}{}^1)_x = D\mathbf{f}_x|_{\mathbf{v}}$ is defined by summing up the differentials with respect to the single variables. We consider the differential with respect to variable y and the summand that we get by selecting monomial m_x (Part (5) of the Definition). This summand itself is a sum obtained by the application of the product rule (Part (6)) to m_x. Note that fully unfolding the product rule means that the base case (Parts (7) and 8)) is applied to one single symbol in m_x, and all other symbols are evaluated at $\mathbf{v}$. We consider the summand $s = eval_{\mathbf{v}}(m_1) \odot D_y y|_{\mathbf{v}} \odot eval_{\mathbf{v}}(m_2)$ that is obtained by evaluating all symbols but y. The differential of y with respect to y is again y, so we get $s = eval_{\mathbf{v}}(m_1) \odot y \odot eval_{\mathbf{v}}(m_2)$. This shows that the summand is evaluated to

$$\begin{aligned} eval_{\mathbf{v}}(eval_{\mathbf{v}}(m_1) \odot y \odot eval_{\mathbf{v}}(m_2)) &= eval_{\mathbf{v}}(m_1) \odot eval_{\mathbf{v}}(y) \odot eval_{\mathbf{v}}(m_2)\\ &= eval_{\mathbf{v}}(m_1 \odot y \odot m_2)\\ &= eval_{\mathbf{v}}(m_x) \ . \end{aligned}$$

Let us now consider $\tau_y = \{y \mapsto g\}$, with $g \in m_y\tau_z$ (where m_y is a monomial of $\mathbf{f}_y$). By induction, there is an i and a summand s' of $(D\mathbf{f}|_{\mathbf{v}}{}^i)_y$ such that evaluating s' and g leads to the same result. We look at $(D\mathbf{f}|_{\mathbf{v}}{}^{i+1})_x = eval_{D\mathbf{f}|_{\mathbf{v}}{}^{i}}(D\mathbf{f}_x|_{\mathbf{v}})$. We consider each summand $s = eval_{\mathbf{v}}(m_1) \odot y \odot eval_{\mathbf{v}}(m_2)$ of $D\mathbf{f}_x|_{\mathbf{v}}$ as in the base case. Evaluating this summand at $D\mathbf{f}|_{\mathbf{v}}{}^{i}$ will evaluate y to the sum $(D\mathbf{f}|_{\mathbf{v}}{}^{i})_y$ containing s'. Using distributivity yields a new sum containing the summand given by evaluating s at the summand s' of $D\mathbf{f}_x|_{\mathbf{v}}{}^{i}$. Using the assumption that s evaluates to g, this evaluates to

$$\begin{aligned} eval_{\mathbf{v}}(eval_{\mathbf{v}}(m_1) \odot s' \odot eval_{\mathbf{v}}(m_2)) &= eval_{\mathbf{v}}(m_1) \odot eval_{\mathbf{v}}(s') \odot eval_{\mathbf{v}}(m_2) \\ &= eval_{\mathbf{v}}(m_1) \odot eval_{\mathbf{v}}(g) \odot eval_{\mathbf{v}}(m_2) \\ &= eval_{\mathbf{v}}(m_1 \odot g \odot m_2) \\ &= eval_{\mathbf{v}}(m\tau_y) \ . \end{aligned}$$

To show $\geq$, we argue that summands of the polynomial defining $(D\mathbf{f}|_{\mathbf{v}}{}^{i})_x$ correspond to substitutions applied to x. For $(D\mathbf{f}|_{\mathbf{v}}{}^{0})_x = id_x$, we can select the substitution $\{x \to x\}$.

We will show that for any $i > 0$ and any summand s of $(D\mathbf{f}_x|_{\mathbf{v}}{}^{i})_x$, there is a monomial m_x of $\mathbf{f}_x$, a substitution τ_y and $g \in m_x\tau_x$ such that s and g evaluate to the same result.

In the base case, note that $(D\mathbf{f}_x|_{\mathbf{v}}{}^{1})_x = D\mathbf{f}_x|_{\mathbf{v}}$ is a sum of the $D_y\mathbf{f}_x|_{\mathbf{v}}$ for all $y \in \mathcal{X}$. Let us fix some y, then $D_y\mathbf{f}_x|_{\mathbf{v}}$ is a sum with the summands corresponding to the monomials of $\mathbf{f}_x$ (Part (5)). If some monomial m_x does not contain y, all unfoldings of the product rule will have 0 as a factor. Analogously, unfolding the product rule such that the differential is applied to a symbol other than y will result in 0. Let us fix an unfolding of the product rule not resulting in 0, and let $m_x = m_1 \odot y \odot m_2$ be the corresponding decomposition of m_x. The corresponding summand of $D_y\mathbf{f}_x|_{\mathbf{v}}$ is $eval_{\mathbf{v}}(m_1) \odot y \odot eval_{\mathbf{v}}(m_2)$. With an argumentation analogous to the one used in the first part of the proof, this evaluates just as the element of $m_x\{y \to y\}$ does, where the substitution is applied to the occurrence of y as in the decomposition.

Now let us consider $(D\mathbf{f}|_{\mathbf{v}}{}^{i+1})_x = eval_{D\mathbf{f}|_{\mathbf{v}}{}^{i}}(D\mathbf{f}_x|_{\mathbf{v}})$. Every summand of $(D\mathbf{f}|_{\mathbf{v}}{}^{i+1})_x$ corresponds to evaluating a summand s of $D\mathbf{f}_x|_{\mathbf{v}}$ at $D\mathbf{f}|_{\mathbf{v}}{}^{i}$. As in the base case, we can assume that s has shape $eval_{\mathbf{v}}(m_1) \odot y \odot eval_{\mathbf{v}}(m_2)$. Evaluating s will result in $eval_{\mathbf{v}}(m_1) \odot (D\mathbf{f}|_{\mathbf{v}}{}^{i})_y \odot eval_{\mathbf{v}}(m_2)$. Using distributivity, we get a large sum in which one single summand corresponds to evaluating s at a summand s' of $(D\mathbf{f}|_{\mathbf{v}}{}^{i})_y$. By induction, there is a monomial m_y of $\mathbf{f}_y$, a substitution τ_z and $h \in m_y\tau_z$ such that s' and h evaluate to the same result. We consider the substitution $\tau_y = \{y \mapsto h\}$, and $g \in m_x\tau_y$, where the substitution applies to the same occurrence of y as in s. Using that s' and h evaluate to the same result, we get

$$\begin{aligned} eval_{\mathbf{v}}(eval_{\mathbf{v}}(m_1) \odot s' \odot eval_{\mathbf{v}}(m_2)) &= eval_{\mathbf{v}}(m_1) \odot eval_{\mathbf{v}}(s') \odot eval_{\mathbf{v}}(m_2) \\ &= eval_{\mathbf{v}}(m_1) \odot eval_{\mathbf{v}}(h) \odot eval_{\mathbf{v}}(m_2) \\ &= eval_{\mathbf{v}}(m_1 \odot h \odot m_2) \\ &= eval_{\mathbf{v}}(g) \ . \end{aligned}$$

◀

C Decorated Derivation Trees

The nodes in the derivation trees from [3] are **decorated**: They are not only labeled by a symbol, but also by the rule that was used to derive the symbol. Let $\mathcal{D}_n(x)$ denote the set of all decorated complete derivation trees of dimension at most n. One derivation tree as defined in our setting might correspond to several derivation trees with different additional labels. We obtain each tree in $\mathcal{C}_n(x)$ by projecting all labels of a tree in $\mathcal{D}_n(x)$ to the first component, and every tree in $\mathcal{D}_n(x)$ can be projected to a tree in $\mathcal{C}_n(x)$. Since the *yield* function ignores the additional labels and since we assume idempotence, we end up with the same result if we sum up the yields of all undecorated trees:

$$\begin{aligned}\bigoplus_{t\in\mathcal{D}_n(x)} \mathit{yield}(t) &= \bigoplus_{t\in\mathcal{C}_n(x)} \bigoplus_{\substack{t'\in\mathcal{D}_n(x),\\ \mathit{project}(t')=t}} \mathit{yield}(t') \\ &= \bigoplus_{t\in\mathcal{C}_n(x)} \bigoplus_{\substack{t'\in\mathcal{D}_n(x),\\ \mathit{project}(t')=t}} \mathit{yield}(t) \\ &= \bigoplus_{t\in\mathcal{C}_n(x)} \mathit{yield}(t)\ .\end{aligned}$$

D The Non-Idempotent Case

One may ask whether the Munchausen sequence also converges to the least fixed point in the case when the underlying semiring is not idempotent. The proof of Theorem 6 does not hold in the non-idempotent case: The trees of dimension lower than 2^{n+1} are summed up several times. (For example, a tree of dimension $2^n + 1$ may occur in the sum as a list of trees of dimension 1 plugged into a tree of dimension 2^n and as a list of trees of dimension 2^n plugged into a tree of dimension 1). This problem cannot be solved by considering decorated derivation trees as in [3]. Even if we distinguish derivation trees that have the same shape but were created using different rules, the sum might contain multiple occurrences of one decorated derivation tree. Therefore, in the non-idempotent case, the convergence results for Newton iteration do not carry over to Munchausen iteration.

We demonstrate that indeed Munchausen iteration may compute values strictly larger than the least fixed point in the following example.

▶ **Example 17.** Consider the following system of equations over the commutative but not idempotent ω-continuous semiring of natural numbers with infinity $(\mathbb{N} \cup \{\infty\}, +, \cdot, 0, 1)$:

$$x = y \cdot y \qquad\qquad y = z \qquad\qquad z = 2\ .$$

Applying the decomposition into the non-constant and the constant part, we may write it as $\mathbf{x} = \mathbf{f} + \mathbf{a} = (y \cdot y, z, 0) + (0, 0, 2)$. Its linear completion is

$$lc(\mathbf{f}_x) = x + y \cdot y + z \cdot y + y \cdot z \qquad\qquad lc(\mathbf{f}_y) = y + z \qquad\qquad lc(\mathbf{f}_z) = z\ .$$

Evaluating it at the vector of constants $(0, 0, 2)$ yields $\mathit{eval}_{\mathbf{a}}(lc(\mathbf{f})) = (0, 2, 2)$. Plugging in the linear completion into itself to obtain the 1st Munchausen approximant yields

$$\begin{aligned}{\beta^{(1)}}_x &= (x + y \cdot y + z \cdot y + y \cdot z) + (y + z) \cdot (y + z) + z \cdot (y + z) + (y + z) \cdot z \\ {\beta^{(1)}}_y &= (y + z) + z \\ {\beta^{(1)}}_z &= z\ .\end{aligned}$$

We get $\beta(\mathbf{a})^{(1)} = (12, 4, 2)$, which is already strictly larger than the least fixed point $(4, 2, 2)$.