On Sharing, Memoization, and Polynomial Time*

Martin Avanzini and Ugo Dal Lago

Università di Bologna & INRIA, Sophia Antipolis
`martin.avanzini@uibk.ac.at` and `dallago@cs.unibo.it`

Abstract

We study how the adoption of an evaluation mechanism with sharing and memoization impacts the class of functions which can be computed in polynomial time. We first show how a natural cost model in which lookup for an already computed result has no cost is indeed invariant. As a corollary, we then prove that the most general notion of ramified recurrence is sound for polynomial time, this way settling an open problem in implicit computational complexity.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases implicit computational complexity, data-tiering, polynomial time

Digital Object Identifier 10.4230/LIPIcs.STACS.2015.62

1 Introduction

Traditionally, complexity classes are defined by giving bounds on the amount of resources algorithms are allowed to use while solving problems. This, in principle, leaves open the task of understanding the *structure* of complexity classes. As an example, a given class of functions is not necessarily closed under composition or, more interestingly, under various forms of recursion. When the class under consideration is not too large, say close enough to the class of *polytime computable functions*, closure under recursion does not hold: iterating over an efficiently computable function is not necessarily efficiently computable, e.g. when the iterated function grows more than linearly. In other words, characterizing complexity classes by purely recursion-theoretical means is non-trivial.

In the past twenty years, this challenge has been successfully tackled, by giving *restricted* forms of recursion for which not only certain complexity classes are closed, but which *precisely* generate the class. This has been proved for classes like PTime, PSpace, the polynomial hierarchy PH, or even smaller ones like NC. A particularly fruitful direction has been the one initiated by Bellantoni and Cook, and independently by Leivant, which consists in restricting the primitive recursive scheme by making it *predicative*, thus forbidding those nested recursive definitions which lead outside the classes cited above. Once this is settled, one can tune the obtained scheme by either adding features (e.g. parameter substitutions) or further restricting the scheme (e.g. by way of linearization).

Something a bit disappointing in this field is that the expressive power of the simplest (and most general) form of predicative recurrence, namely *simultaneous* recurrence on *generic algebras* is unknown. If algebras are restricted to be *string* algebras, or if recursion is not simultaneous, soundness for polynomial time computation is known to hold [21, 16]. The two soundness results are obtained by quite different means, however: in presence of trees, one is

* This work was partially supported by FWF project number J 3563 and by French ANR project Elica ANR-14-CE25-0005.

forced to handle *sharing* [16] of common sub-expressions, while simultaneous definitions by recursion requires a form of *memoization* [21].

In this paper, we show that sharing and memoization can indeed be reconciled, and we exploit both to give a new invariant time cost model for the evaluation of rewrite systems. This paves the way towards polytime soundness for simultaneous predicative recursion on generic algebras, thus solving the open problem we were mentioning. More precisely, with the present paper we make the following contributions:

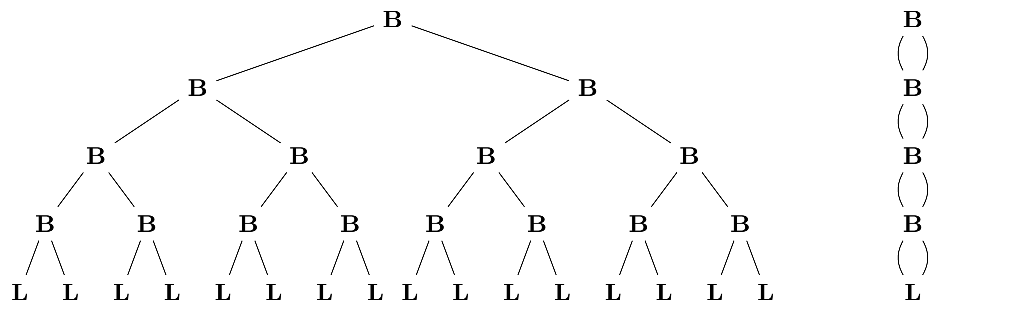
1. We define a simple functional programming language. The domain of the defined functions is a free algebra formed from constructors. Hence we can deal with functions over strings, lists, but also trees (see Section 3). We then extend the underlying rewriting based semantics with *memoization*, i.e. intermediate results are automatically tabulated to avoid expensive re-computation (Section 4). As standard for functional programming languages such as Haskell or OCaml, data is stored in a *heap*, facilitating *sharing* of common sub-expression. To measure the *runtime* of such programs, we employ a novel cost model, called *memoized runtime complexity*, where each function application counts one time unit, but lookups of tabulated calls do not have to be accounted.
2. Our *invariance theorem* (see Theorem 17) relates, within a polynomial overhead, the memoized runtime complexity of programs to the cost of implementing the defined functions on a classical model of computation, e.g. *Turing* or *random access machines*. The invariance theorem thus confirms that our cost model truthfully represents the computational complexity of the defined function.
3. We extend upon Leivant's notion of *ramified recursive functions* [20] by allowing definitions by *generalised ramified simultaneous recurrence* (*GRSR* for short). We show that the resulting class of functions, defined over arbitrary free algebras have, when implemented as programs, polynomial memoized runtime complexity (see Theorem 21). By our invariance theorem, the function algebra is sound for polynomial time, and consequently GRSR characterizes the class of polytime computable functions.

An extended version of this paper with more details, including all proofs, is also available [4].

1.1 Related Work

That predicative recursion *on strings* is sound for polynomial time, even in presence of simultaneous recursive definitions, is known for a long time [9]. Variations of predicative recursion have been later considered and proved to characterize classes like PH [10], PSpace [23], ExpTime [3] or NC [12]. Predicative recursion on trees has been claimed to be sound for polynomial time in the original paper by Leivant [20], the long version of which only deals with strings [21]. After fifteen years, the non-simultaneous case has been settled by the second author with Martini and Zorzi [16]; their proof, however, relies on an ad-hoc, infinitary, notion of graph rewriting. Recently, ramification has been studied in the context of a simply-typed λ-calculus in an unpublished manuscript [17]; the authors claim that a form of ramified recurrence on trees captures polynomial time; this, again, does not take simultaneous recursion into account.

The formalism presented here is partly inspired by the work of Hoffmann [19], where sharing and memoization are shown to work well together in the realm of term graph rewriting. The proposed machinery, although powerful, is unnecessarily complicated for our purposes. Speaking in Hoffmann's terms, our results require a form of full memoization, which *is* definable in Hoffmann's system. However, most crucially for our concerns, it is unclear how the overall system incorporating full memoization can be implemented efficiently, if at all.



(a) Explicit tree representation. **(b)** Compact DAG.

Figure 1 Complete Binary Tree of Height Four, as Computed by $\texttt{tree}(\mathbf{S}^4(\mathbf{0}))$.

2 The Need for Sharing and Memoisation

This Section is an informal, example-driven, introduction to ramified recursive definitions and their complexity. Our objective is to convince the reader that those definitions do *not* give rise to polynomial time computations if naively evaluated, and that sharing and memoization are *both* necessary to avoid exponential blowups.

In Leivant's system [21], functions and variables are equipped with a *tier*. Composition must preserve tiers and, crucially, in a function defined by primitive recursion the tier of the recurrence parameter must be higher than the tier of the recursive call. This form of *ramification* of functions effectively tames primitive recursion, resulting in a characterisation of the class of *polytime computable functions*.

Of course, ramification also controls the growth rate of functions. However, as soon as we switch from strings to a domain where tree structures are definable, this control is apparently lost. For illustration, consider the following definition:

$$\texttt{tree}(\mathbf{0}) = \mathbf{L} \qquad \texttt{tree}(\mathbf{S}(n)) = \texttt{br}(\texttt{tree}(n)) \qquad \texttt{br}(t) = \mathbf{B}(t, t)\ .$$

The function $\texttt{tree}$ is defined by primitive recursion, essentially from basic functions. As a consequence, it is easily seen to be ramified in the sense of Leivant. Even though the number of recursive steps is linear in the input, the result of $\texttt{tree}(\mathbf{S}^n(\mathbf{0}))$ is the complete binary tree of height n. As thus the length of the output is exponential in the one of its input, there is, at least apparently, little hope to prove $\texttt{tree}$ a polytime function. The way out is *sharing*: the complete binary tree of height n can be compactly represented as a *directed acyclic graph* (*DAG* for short) of linear size (see Figure 1). Indeed, using the compact DAG representation it is easy to see that the function $\texttt{tree}$ is computable in polynomial time. This is the starting point of [16], in which general ramified recurrence is proved sound for polynomial time. A crucial observation here is that *not only* the output's size, but also the total amount of work can be kept under control, thanks to the fact that evaluating a primitive recursive definition on a compactly represented input can be done by constructing an isomorphic DAG of recursive calls.

This does not scale up to *simultaneous* ramified recurrence. The following example computes the genealogical tree associated with *Fibonacci's rabbit problem* for $n \in \mathbb{N}$ generations. Rabbits come in pairs. After one generation, each *baby* rabbit pair ($\mathbf{N}$) matures. In each

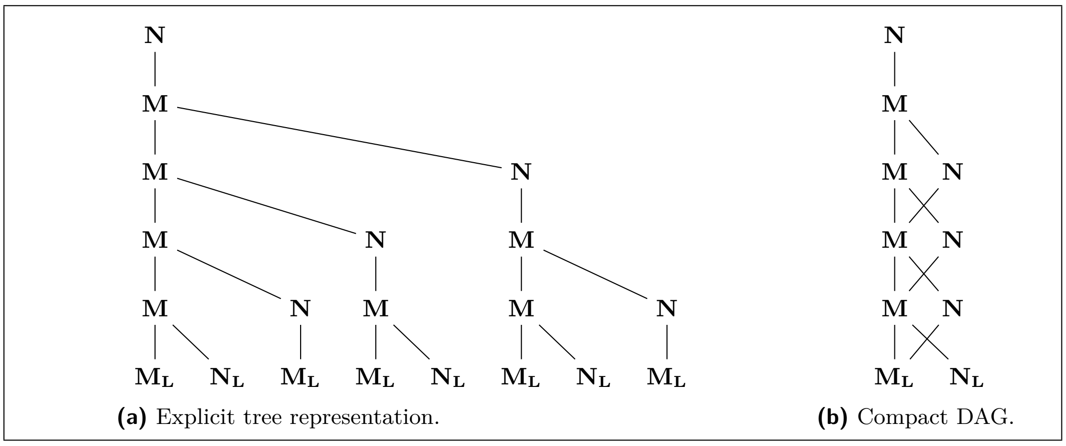


Figure 2 Genealogical Rabbit Tree up to the Sixth Generation, as Computed by $\texttt{rabbits}(\mathbf{S}^6(\mathbf{0}))$.

generation, an *adult* rabbit pair ($\mathbf{M}$) bears one pair of babies.

$$\begin{aligned} \texttt{rabbits}(\mathbf{0}) &= \mathbf{N_L} & \texttt{a}(\mathbf{0}) &= \mathbf{M_L} & \texttt{b}(\mathbf{0}) &= \mathbf{N_L} \\ \texttt{rabbits}(\mathbf{S}(n)) &= \texttt{b}(n) & \texttt{a}(\mathbf{S}(n)) &= \mathbf{M}(\texttt{a}(n), \texttt{b}(n)) & \texttt{b}(\mathbf{S}(n)) &= \mathbf{N}(\texttt{a}(n)) \ . \end{aligned}$$

The function `rabbits` is obtained by case analysis from the functions `a` and `b`, which are defined by *simultaneous* primitive recursion: the former recursively calls itself *and* the latter, while the latter makes a recursive call to the former. The output of $\texttt{rabbits}(\mathbf{S}^n(\mathbf{0}))$ is tightly related to the sequence of Fibonacci numbers: the number of nodes at depth i is given by the i^{th} Fibonacci number. Hence the output tree has exponential size in n but, again, can be represented compactly (see Figure 2). This does not suffice for our purposes, however. In presence of simultaneous definitions, indeed, avoiding re-computation of previously computed values becomes more difficult, the trick described above does not work, and the key idea towards that is the use of *memoization*.

What we prove in this paper is precisely that sharing and memoization can indeed be made to work together, and that they together allow to prove polytime soundness for all ramified recursive functions, also in presence of tree algebras and simultaneous definitions.

3 Preliminaries

General Ramified Simultaneous Recurrence

Let $\mathbb{A}$ denote a *(finite and untyped) signatures* of *constructors* $\mathbf{c}_1, \dots, \mathbf{c}_k$, each equipped with an arity $\mathsf{ar}(\mathbf{c}_i)$. In the following, the set of terms $\mathcal{T}(\mathbb{A})$ over the signature $\mathbb{A}$, defined as usual, is also denoted by $\mathbb{A}$ if this does not create ambiguities. We are interested in total functions from $\mathbb{A}^n = \underbrace{\mathbb{A} \times \ldots \times \mathbb{A}}_{n \text{ times}}$ to $\mathbb{A}$.

▶ **Definition 1.** The following are so-called *basic functions*:
- For each constructor $\mathbf{c}$, the *constructor function* $\mathtt{f}_{\mathbf{c}} : \mathbb{A}^{\mathsf{ar}(\mathbf{c})} \to \mathbb{A}$ for $\mathbf{c}$, defined as follows: $\mathtt{f}_{\mathbf{c}}(x_1, \dots, x_{\mathsf{ar}(\mathbf{c})}) = \mathbf{c}(x_1, \dots, x_{\mathsf{ar}(\mathbf{c})})$
- For each $1 \leq n \leq m$, the (m,n)-*projection function* $\Pi^m_n : \mathbb{A}^m \to \mathbb{A}$ defined as follows: $\Pi^m_n(x_1 \dots, x_m) = x_n$.

$$\frac{}{\mathtt{f}_{\mathbf{c}} \rhd \mathbb{A}_n^{\mathsf{ar}(\mathbf{c})} \to \mathbb{A}_n} \qquad \frac{}{\Pi^n_m \rhd \mathbb{A}_{p_1} \times \ldots \times \mathbb{A}_{p_m} \to \mathbb{A}_{p_n}} \qquad \frac{\mathtt{f}_i \rhd \mathbb{A}_p^{\mathsf{ar}(\mathbf{c}_i)} \times \mathbf{A} \to \mathbb{A}_m}{case(\{\mathtt{f}_i\}_{1\leq i\leq k}) \rhd \mathbb{A}_p \times \mathbf{A} \to \mathbb{A}_m}$$

$$\frac{\mathtt{f} \rhd \mathbb{A}_{p_1} \times \ldots \times \mathbb{A}_{p_n} \to \mathbb{A}_m \quad \mathtt{g}_i \rhd \mathbf{A} \to \mathbb{A}_{p_i}}{\mathtt{f} \circ (\mathtt{g}_1, \ldots, \mathtt{g}_n) \rhd \mathbf{A} \to \mathbb{A}_m} \qquad \frac{\mathtt{f}_i^j \rhd \mathbb{A}_p^{\mathsf{ar}(\mathbf{c}_i)} \times \mathbb{A}_m^{n\cdot\mathsf{ar}(\mathbf{c}_i)} \times \mathbf{A} \to \mathbb{A}_m \quad p > m}{simrec(\{\mathtt{f}_i^j\}_{1\leq i\leq k, 1\leq j\leq n}) \rhd \mathbb{A}_p \times \mathbf{A} \to \mathbb{A}_m}$$

Figure 3 Tiering as a Formal System.

▶ **Definition 2.**

- Given a function $\mathtt{f} : \mathbb{A}^n \to \mathbb{A}$ and n functions $\mathtt{g}_1, \ldots, \mathtt{g}_n$, all of them from $\mathbb{A}^m$ to $\mathbb{A}$, the *composition* $\mathtt{h} = \mathtt{f} \circ (\mathtt{g}_1, \ldots, \mathtt{g}_n)$ is a function from $\mathbb{A}^m$ to $\mathbb{A}$ defined as follows: $\mathtt{h}(\vec{x}) = \mathtt{f}(\mathtt{g}_1(\vec{x}), \ldots, \mathtt{g}_n(\vec{x}))$.
- Suppose given the functions $\mathtt{f}_i$ where $1 \leq i \leq k$ such that for some m, $\mathtt{f}_i : \mathbb{A}^{\mathsf{ar}(\mathbf{c}_i)} \times \mathbb{A}^n \to \mathbb{A}$. Then the function $\mathtt{g} = case(\{\mathtt{f}_i\}_{1\leq i\leq k})$ defined by *case distinction* from $\{\mathtt{f}_i\}_{1\leq i\leq k}$ is a function from $\mathbb{A} \times \mathbb{A}^n$ to $\mathbb{A}$ defined as follows: $\mathtt{g}(\mathbf{c}_i(\vec{x}), \vec{y}) = \mathtt{f}_i(\vec{x}, \vec{y})$.
- Suppose given the functions $\mathtt{f}_i^j$, where $1 \leq i \leq k$ and $1 \leq j \leq n$, such that for some m, $\mathtt{f}_i^j : \mathbb{A}^{\mathsf{ar}(\mathbf{c}_i)} \times \mathbb{A}^{n\cdot\mathsf{ar}(\mathbf{c}_i)} \times \mathbb{A}^m \to \mathbb{A}$. The functions $\{\mathtt{g}_j\}_{1\leq j\leq n} = simrec(\{\mathtt{f}_i^j\}_{1\leq i\leq k, 1\leq j\leq n})$ defined by *simultaneous primitive recursion* from $\{\mathtt{f}_i^j\}_{1\leq i\leq k, 1\leq j\leq n}$ are all functions from $\mathbb{A} \times \mathbb{A}^m$ to $\mathbb{A}$ such that for $\vec{x} = x_1, \ldots, x_{\mathsf{ar}(\mathbf{c}_i)}$,

$$\mathtt{g}_j(\mathbf{c}_i(\vec{x}), \vec{y}) = \mathtt{f}_i^j(\vec{x}, \mathtt{g}_1(x_1, \vec{y}), \ldots, \mathtt{g}_1(x_{\mathsf{ar}(\mathbf{c}_i)}, \vec{y}), \ldots, \mathtt{g}_n(x_1, \vec{y}), \ldots, \mathtt{g}_n(x_{\mathsf{ar}(\mathbf{c}_i)}, \vec{y}), \vec{y}) \ .$$

We denote by SimRec($\mathbb{A}$) the class of *simultaneous recursive functions over* $\mathbb{A}$, defined as the smallest class containing the basic functions of Definition 1 and that is closed under the schemes of Definition 2.

Tiering, the central notion underlying Leivant's definition of *ramified recurrence*, consists in attributing *tiers* to inputs and outputs of some functions among the ones constructed as above, with the goal of isolating the polytime computable ones. Roughly speaking, the role of tiers is to single out "a copy" of the signature by a level: this level permits to control the recursion nesting. Tiering can be given as a formal system, in which judgments have the form $\mathtt{f} \rhd \mathbb{A}_{p_1} \times \ldots \times \mathbb{A}_{p_{\mathsf{ar}(\mathtt{f})}} \to \mathbb{A}_m$ for $p_1, \ldots, p_{\mathsf{ar}(\mathtt{f})}, m$ natural numbers and $\mathtt{f} \in$ SimRec($\mathbb{A}$). The system is defined in Figure 3, where $\mathbf{A}$ denotes the expression $\mathbb{A}_{q_1} \times \ldots \times \mathbb{A}_{q_k}$ for some $q_1, \ldots, q_k \in \mathbb{N}$. Notice that composition preserves tiers. Moreover, recursion is allowed only on inputs of tier higher than the tier of the function (in the case $\mathtt{f} = simrec(\{\mathtt{f}_i^j\}_{1\leq i\leq k, 1\leq j\leq n})$, we require $p > m$).

▶ **Definition 3.** We call a function $\mathtt{f} \in$ SimRec($\mathbb{A}$) definable by *general ramified simultaneous recurrence* (*GRSR* for short) if $\mathtt{f} \rhd \mathbb{A}_{p_1} \times \ldots \times \mathbb{A}_{p_{\mathsf{ar}(\mathtt{f})}} \to \mathbb{A}_m$ holds.

▶ Remark. Consider the *word algebra* $\mathbb{W} = \{\epsilon, \mathbf{a}, \mathbf{b}\}$ consisting of a constant ϵ and two unary constructors $\mathbf{a}$ and $\mathbf{b}$, which is in bijective correspondence to the set of binary words. Then the functions definable by ramified simultaneous recurrence over $\mathbb{W}$ includes the ramified recursive functions from Leivant [21], and consequently all polytime computable functions.

▶ **Example 4.**

1. Consider $\mathbb{N} := \{\mathbf{0}, \mathbf{S}\}$ with $\mathsf{ar}(\mathbf{0}) = 0$ and $\mathsf{ar}(\mathbf{S}) = 1$, which is in bijective correspondence to the set of natural numbers. We can define addition $\mathtt{add} : \mathbb{N}_i \times \mathbb{N}_j \to \mathbb{N}_j$ for $i > j$, by

$$\mathtt{add}(\mathbf{0}, y) = \Pi^1_1(y) = y \qquad \mathtt{add}(\mathbf{S}(x), y) = (\mathtt{f}_{\mathbf{S}} \circ \Pi^3_2)(x, \mathtt{add}(x, y), y) = \mathbf{S}(\mathtt{add}(x, y)) \ ,$$

using general simultaneous ramified recursion, i.e. $\{\mathtt{add}\} = simrec(\{\{\Pi^1_1, \mathtt{f}_{\mathbf{S}} \circ \Pi^3_2\}\})$.

$$\frac{\mathtt{f} \in \mathcal{F} \quad t_i \downarrow v_i \quad f(v_1,\dots,v_k) \downarrow v}{f(t_1,\dots,t_k) \downarrow v} \qquad \frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1,\dots,t_k) \downarrow \mathbf{c}(v_1,\dots,v_k)}$$

$$\frac{f(p_1,\dots,p_k) \to r \in \mathcal{R} \quad \forall i.\ p_i\sigma = v_i \quad r\sigma \downarrow v}{f(v_1,\dots,v_k) \downarrow v}$$

Figure 4 Operational Semantics for Program $(\mathcal{F}, \mathcal{C}, \mathcal{R})$.

2. Let $\mathbb{T} := \{\mathbf{N_L}, \mathbf{M_L}, \mathbf{N}, \mathbf{M}\}$, where $\mathsf{ar}(\mathbf{N_L}) = \mathsf{ar}(\mathbf{M_L}) = 0$, $\mathsf{ar}(\mathbf{N}) = 1$ and $\mathsf{ar}(\mathbf{M}) = 2$. Then we can define the functions $\mathtt{rabbits} : \mathbb{N}_i \to \mathbb{T}_j$ for $i > j$ from Section 2 by composition from the following two functions, defined by simultaneous ramified recurrence.

$$\begin{aligned} &\mathtt{a}(\mathbf{0}) = \mathbf{M_L} && \mathtt{a}(\mathbf{S}(n)) = (\mathtt{f}_\mathbf{M} \circ (\Pi^3_2, \Pi^3_3))\,(n, \mathtt{a}(n), \mathtt{b}(n)) = \mathbf{M}(\mathtt{a}(n), \mathtt{b}(n)) \\ &\mathtt{b}(\mathbf{0}) = \mathbf{N_L} && \mathtt{b}(\mathbf{S}(n)) = (\mathtt{f}_\mathbf{N} \circ \Pi^3_3)\,(n, \mathtt{a}(n), \mathtt{b}(n)) = \mathbf{N}(\mathtt{a}(n))\ . \end{aligned}$$

3. We can define a function $\#\mathtt{leafs} : \mathbb{T} \to \mathbb{N}$ by simultaneous primitive recursion which counts the number of leafs in $\mathbb{T}$-trees as follows.

$$\begin{aligned} &\#\mathtt{leafs}(\mathbf{N_L}) = \mathbf{S}(\mathbf{0}) && \#\mathtt{leafs}(\mathbf{M_L}) = \mathbf{S}(\mathbf{0}) \\ &\#\mathtt{leafs}(\mathbf{N}(t)) = \#\mathtt{leafs}(t) && \#\mathtt{leafs}(\mathbf{M}(l, r)) = \mathtt{add}(\#\mathtt{leafs}(l), \#\mathtt{leafs}(r))\ . \end{aligned}$$

However, this function *cannot* be ramified, since $\mathtt{add}$ in the last equation requires different tiers. Indeed, having a ramified recursive function $\#\mathtt{leafs} : \mathbb{T}_i \to \mathbb{N}_1$ (for some $i > 1$) defined as above would allow us to ramify $\mathtt{fib} = \#\mathtt{leafs} \circ \mathtt{rabbits}$ which on input n computes the n^{th} Fibonacci number, and is thus an exponential function.

Computational Model, Syntax and Semantics

We introduce a simple, *rewriting based*, notion of program for computing functions over term algebras. Let $\mathcal{V}$ denote a set of *variables*. Terms over a signature $\mathcal{F}$ that include variables from $\mathcal{V}$ are denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is called *linear* if each variable occurs at most once in t. The set of *subterms* $\mathsf{STs}(t)$ of a term t is defined by $\mathsf{STs}(t) := \{t\}$ if $t \in \mathcal{V}$ and $\mathsf{STs}(t) := \bigcup_{1 \leq i \leq \mathsf{ar}(f)} \mathsf{STs}(t_i) \cup \{t\}$ if $t = f(t_1, \dots, t_{\mathsf{ar}(f)})$. A *substitution*, is a finite mapping σ from variables to terms. By $t\sigma$ we denote the term obtained by replacing in t all variables $x \in \mathsf{dom}(\sigma)$ by $\sigma(x)$.

▶ **Definition 5.** A *program* $\mathtt{P}$ is given as a triple $(\mathcal{F}, \mathcal{C}, \mathcal{R})$ consisting of two disjoint signatures $\mathcal{F}$ and $\mathcal{C}$ of *operation symbols* $f_1, \dots, f_m$ and *constructors* $\mathbf{c}_1, \dots, \mathbf{c}_n$ respectively, and a *finite* set $\mathcal{R}$ of *rules* $l \to r$ over terms $l, r \in \mathcal{T}(\mathcal{F} \cup \mathcal{C}, \mathcal{V})$. For each rule, the *left-hand side* l is of the form $f_i(p_1, \dots, p_k)$ for some i, where the *patterns* p_j consist only of variables and constructors, and all variables occurring in the *right-hand side* r also occur in the left-hand side l.

We keep the program $\mathtt{P} = (\mathcal{F}, \mathcal{C}, \mathcal{R})$ fixed throughout the following. Moreover, we require that $\mathtt{P}$ is *orthogonal*, that is, the following two requirements are met:
1. *left-linearity:* the left-hand sides l of each rule $l \to r \in \mathcal{R}$ is *linear*; and
2. *non-ambiguity:* there are no two rules with overlapping left-hand sides in $\mathcal{R}$.

Orthogonal programs define a class of deterministic first-order functional programs, see e.g. [6]. The domain of the defined functions is the constructor algebra $\mathcal{T}(\mathcal{C})$. Correspondingly, elements of $\mathcal{T}(\mathcal{C})$ are called *values*, which we denote by $v, u, \dots$.

In Figure 4 we present the operational semantics, realizing standard *call-by-value* evaluation order. The statement $t \downarrow v$ means that the term t *reduces* to the value v. We say that $\mathtt{P}$ computes the function $\mathtt{f} : \mathcal{T}(\mathcal{C})^k \to \mathcal{T}(\mathcal{C})$ if there exists an operation $f \in \mathcal{F}$ such that $\mathtt{f}(v_1, \ldots, v_k) = v$ if and only if $f(v_1, \ldots, v_k) \downarrow v$ holds for all inputs $v_i \in \mathcal{T}(\mathcal{C})$.

▶ **Example 6** (Continued from Example 4). The definition of $\mathtt{rabbits}$ from Section 2 can be turned into a program $\mathtt{P_R}$ over constructors of $\mathbb{N}$ and $\mathbb{T}$, by *orienting* the underlying equations from left to right and replacing applications of functions $\mathtt{f} \in \{\mathtt{rabbits}, \mathtt{a}, \mathtt{b}\}$ with corresponding operation symbols $f \in \{\mathit{rabbits}, a, b\}$. For instance, concerning the function $\mathtt{a}$, the defining equations are turned into $a(\mathbf{0}) \to \mathbf{M_L}$ and $a(\mathbf{S}(n)) \to \mathbf{M}(a(n), b(n))$.

The example hints at a systematic construction of programs $\mathtt{P_f}$ computing functions $\mathtt{f} \in \textsc{SimRec}(\mathbb{A})$, which can be made precise [4].

Term Graphs

We borrow key concepts from *term graph rewriting* (see e.g. the survey of Plump [25] for an overview) and follow the presentation of Barendregt et al. [8]. A *term graph* T over a signature $\mathcal{F}$ is a *directed acyclic graph* whose nodes are labeled by symbols in $\mathcal{F} \cup \mathcal{V}$, and where outgoing edges are ordered. Formally, T is a triple $(N, \mathsf{suc}, \mathsf{lab})$ consisting of *nodes* N, a *successors function* $\mathsf{suc} : N \to N^*$ and a *labeling function* $\mathsf{lab} : N \to \mathcal{F} \cup \mathcal{V}$. We require that term graphs are *compatible* with $\mathcal{F}$, in the sense that for each node $o \in N$, if $\mathsf{lab}_T(o) = f \in \mathcal{F}$ then $\mathsf{suc}_T(o) = [o_1, \ldots, o_{\mathsf{ar}(f)}]$ and otherwise, if $\mathsf{lab}_T(o) = x \in \mathcal{V}$, $\mathsf{suc}_T(o) = [\,]$. In the former case, we also write $T(o) = f(o_1, \ldots, o_{\mathsf{ar}(f)})$, the latter case is denoted by $T(o) = x$. We define the *successor relation* $\rightharpoonup_T$ on nodes in T such that $o \rightharpoonup_T p$ holds iff p occurs in $\mathsf{suc}(o)$, if p occurs at the i^{th} position we also write $o \overset{i}{\rightharpoonup}_T p$. Throughout the following, we consider only *acyclic* term graphs, that is, when $\rightharpoonup_T$ is acyclic. Hence the *unfolding* $[o]_T$ of T *at* node o, defined by $[o]_T := x$ if $T(o) = x \in \mathcal{V}$, and otherwise $[o]_T := f([o_1]_T, \ldots, [o_k]_T)$ where $T(o) = f(o_1, \ldots, o_k)$, results in a finite term. We called the term graph T *rooted* if there exists a unique node o, the *root* of T, with $o \rightharpoonup_T^* p$ for every $p \in N$. We denote by $T{\downharpoonright}o$ the *sub-graph* of T *rooted* at o. Consider a symbol $f \in \mathcal{F}$ and nodes $\{o_1, \ldots, o_{\mathsf{ar}(f)}\} \subseteq N$ of T. The extension S of T by a fresh node $o_f \notin N$ with $S(o_f) = f(o_1, \ldots, o_{\mathsf{ar}(f)})$ is denoted by $T \uplus \{o_f \mapsto f(o_1, \ldots, o_{\mathsf{ar}(f)})\}$. We write $f(T{\downharpoonright}o_1, \ldots, T{\downharpoonright}o_{\mathsf{ar}(f)})$ for the term graph $S{\downharpoonright}o_f$.

For two rooted term graphs $T = (N_T, \mathsf{suc}_T, \mathsf{lab}_T)$ and $S = (N_S, \mathsf{suc}_S, \mathsf{lab}_S)$, a mapping $m : N_T \to N_S$ is called *morphic* in $o \in N_T$ if (i) $\mathsf{lab}_T(o) = \mathsf{lab}_S(m(o))$ and (ii) $o \overset{i}{\rightharpoonup}_T p$ implies $m(o) \overset{i}{\rightharpoonup}_S m(p)$ for all appropriate i. A *homomorphism* from T to S is a mapping $m : N_T \to N_S$ that (i) maps the root of T to the root of S and that (ii) is morphic in all nodes $o \in N_T$ not labeled by a variable. We write $T \geqslant_m S$ to indicate that m is, possibly an extension of, a homomorphism from T to S.

Every term t is trivially representable as a *canonical tree* $\triangle(t)$ unfolding to t, using a fresh node for each occurrence of a subterm in t. For t a linear term, to each variable x in t we can associate a *unique node* in $\triangle(t)$ labeled by x, which we denote by o_x. The following proposition relates matching on terms and homomorphisms on trees. It essentially relies on the imposed linearity condition.

▶ **Proposition 7** (Matching on Graphs). *Let t be a linear term, T be a term graph and let o be a node of T.*

1. *If $\triangle(t) \geqslant_m T{\downharpoonright}o$ then there exists a substitution σ such that $t\sigma = [o]_T$.*
2. *Vice versa, if $t\sigma = [o]_T$ holds for some substitution σ then there exists a homomorphism $\triangle(t) \geqslant_m T{\downharpoonright}o$.*

$$\frac{f \in \mathcal{F} \quad (C_{i-1}, t_i) \Downarrow_{n_i} (C_i, v_i) \quad (C_k, f(v_1,\dots,v_k)) \Downarrow_n (C_{k+1}, v) \quad m = n + \sum_{i=1}^{k} n_i}{(C_0, f(t_1,\dots,t_k)) \Downarrow_m (C_{k+1}, v)}\ (\texttt{Split})$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad (C_{i-1}, t_i) \Downarrow_{n_i} (C_i, v_i) \quad m = \sum_{i=1}^{k} n_i}{(C_0, \mathbf{c}(t_1,\dots,t_k)) \Downarrow_m (C_k, \mathbf{c}(v_1,\dots,v_k))}\ (\texttt{Con}) \qquad \frac{(f(v_1,\dots,v_k), v) \in C}{(C, f(v_1,\dots,v_k)) \Downarrow_0 (C, v)}\ (\texttt{Read})$$

$$\frac{(f(v_1,\dots,v_k), v) \notin C \quad f(p_1,\dots,p_k) \to r \in \mathcal{R} \quad \forall i.\ p_i\sigma = v_i \quad (C, r\sigma) \Downarrow_m (D, v)}{(C, f(v_1,\dots,v_k)) \Downarrow_{m+1} (D \cup \{(f(v_1,\dots,v_k), v)\}, v)}\ (\texttt{Update})$$

Figure 5 Cost Annotated Operational Semantics with Memoization for Program $(\mathcal{F}, \mathcal{C}, \mathcal{R})$.

Here, the substitution σ and homomorphism m satisfy $\sigma(x) = [m(o_x)]_T$ for all variables x in t.

4 Memoization and Sharing, Formally

To implement *memoization*, we make use of a *cache* C which stores results of intermediate functions calls. A *cache* C is modeled as a set of tuples $(f(v_1,\dots,v_{\mathsf{ar}(f)}), v)$, where $f \in \mathcal{F}$ and $v_1,\dots,v_{\mathsf{ar}(f)}$ as well as v are values.

Figure 5 collects the *memoizing operational semantics* with respect to the program $\mathtt{P} = (\mathcal{F}, \mathcal{C}, \mathcal{R})$. Here, a statement $(C, t) \Downarrow_m (D, v)$ means that starting with a cache C, the term t *reduces* to the value v with updated cache D. The natural number m indicates the *cost* of this reduction. The definition is along the lines of the standard semantics (Figure 4), carrying the cache throughout the reduction of the given term. The last rule of Figure 4 is split into two rules $(\texttt{Read})$ and $(\texttt{Update})$. The former performs a read from the cache, the latter the reduction in case the corresponding function call is not tabulated, updating the cache with the computed result. Notice that in the semantics, a read is attributed zero cost, whereas an update is accounted with a cost of one. Consequently the cost m in $(C, t) \Downarrow_m (D, v)$ refers to the number of non-tabulated function applications.

▶ **Lemma 8.** *We have $(\varnothing, t) \Downarrow_m (C, v)$ for some $m \in \mathbb{N}$ and cache C if and only if $t \downarrow v$.*

The lemma confirms that the call-by-value semantics of Section 3 is correctly implemented by the memoizing semantics. To tame the growth rate of values, we define *small-step semantics* corresponding to the memoizing semantics, facilitating sharing of common sub-expressions.

Small-Step Semantics with Memoization and Sharing

To incorporate sharing, we extend the pairs (C, t) by a *heap*, and allow *references* to the heap both in terms and in caches. Let Loc denote a countably infinite set of *locations*. We overload the notion of *value*, and define *expressions* e and *(evaluation) contexts* E according to the following grammar:

$$\begin{aligned} v &:= \ell \mid \mathbf{c}(v_1,\dots,v_k); \\ e &:= \ell \mid \langle f(\ell_1,\dots,\ell_k), e\rangle \mid f(e_1,\dots,e_k) \mid \mathbf{c}(e_1,\dots,e_k); \\ E &:= \Box \mid \langle f(\ell_1,\dots,\ell_k), E\rangle \mid f(\ell_1,\dots,\ell_{i-1}, E, e_{i+1},\dots,e_k) \mid \mathbf{c}(\ell_1,\dots,\ell_{i-1}, E, e_{i+1},\dots,e_k). \end{aligned}$$

Here, $\ell_1,\dots,\ell_k,\ell \in \mathsf{Loc}$, $f \in \mathcal{F}$ and $\mathbf{c} \in \mathcal{C}$ are k-ary symbols. An expression is a term including references to values that will be stored on the heap. The additional construct $\langle f(\ell_1,\dots,\ell_k), e\rangle$ indicates that the partially evaluated expression e descends from a call

$$\dfrac{\begin{array}{c}(\boldsymbol{f}(\ell_1,\dots,\ell_k),\ell) \notin D \quad \boldsymbol{f}(p_1,\dots,p_k) \to r \in \mathcal{R} \quad T := \triangle(\boldsymbol{f}(p_1,\dots,p_k)) \\ T \mathrel{\dot{\geqslant}}_m \boldsymbol{f}(H{\downharpoonleft}\ell_1,\dots,H{\downharpoonleft}\ell_k) \quad \sigma_m := \{x \mapsto m(\ell_x) \mid \ell_x \in \mathsf{Loc},\ T(\ell_x) = x \in \mathcal{V}\}\end{array}}{(D,H,E[\boldsymbol{f}(\ell_1,\dots,\ell_k)]) \to_{\mathtt{R}} (D,H,E[\langle \boldsymbol{f}(\ell_1,\dots,\ell_k), r\sigma_m\rangle])}\ (\mathtt{apply})$$

$$\dfrac{(\boldsymbol{f}(\ell_1,\dots,\ell_k),\ell) \in D}{(D,H,E[\boldsymbol{f}(\ell_1,\dots,\ell_k)]) \to_{\mathtt{r}} (D,H,E[\ell])}\ (\mathtt{read})$$

$$\dfrac{}{(D,H,E[\langle \boldsymbol{f}(\ell_1,\dots,\ell_k),\ell\rangle]) \to_{\mathtt{s}} (D \cup \{(\boldsymbol{f}(\ell_1,\dots,\ell_k),\ell)\},H,E[\ell])}\ (\mathtt{store})$$

$$\dfrac{(H',\ell) = \mathsf{merge}(H,\mathbf{c}(\ell_1,\dots,\ell_k))}{(D,H,E[\mathbf{c}(\ell_1,\dots,\ell_k)]) \to_{\mathtt{m}} (D,H',E[\ell])}\ (\mathtt{merge})$$

Figure 6 Small Step Semantics with Memoization and Sharing for Program $(\mathcal{F},\mathcal{C},\mathcal{R})$.

$\boldsymbol{f}(v_1,\dots,v_k)$, with arguments v_i stored at location ℓ_i on the heap. A context E is an expression with a unique *hole*, denoted as $\Box$, where all sub-expression to the left of the hole are references pointing to values. This syntactic restriction is used to implement a *left-to-right*, *call-by-value* evaluation order. We denote by $E[e]$ the expression obtained by replacing the hole in E by e.

A *configuration* is a triple (D,H,e) consisting of a *cache* D, *heap* H and expression e. Unlike before, the cache D consists of pairs of the form $(\boldsymbol{f}(\ell_1,\dots,\ell_k),\ell)$ where instead of values, we store references $\ell_1,\dots,\ell_k,\ell$ pointing to the heap. The heap H is represented as a (multi-rooted) term graph H with nodes in Loc and constructors $\mathcal{C}$ as labels. If ℓ is a node of H, then we say that H stores at location ℓ the value $[\ell]_H$ obtained by unfolding H starting from location ℓ. We keep the heap in a *maximally shared* form, that is, $H(\ell_a) = \mathbf{c}(\ell_1,\dots,\ell_k) = H(\ell_b)$ implies $\ell_a = \ell_b$ for two locations ℓ_a,ℓ_b of H. Thus crucially, values are stored once only, by the following lemma.

▶ **Lemma 9.** *Let H be a maximally shared heap with locations ℓ_1,ℓ_2. If $[\ell_1]_H = [\ell_2]_H$ then $\ell_1 = \ell_2$.*

The operation $\mathsf{merge}(H,\mathbf{c}(\ell_1,\dots,\ell_k))$, defined as follows, is used to extend the heap H with a constructor $\mathbf{c}$ whose arguments point to $\ell_1,\dots,\ell_k$, retaining maximal sharing. Let ℓ_f be the first location not occurring in the nodes N of H (with respect to an arbitrary, but fixed enumeration on Loc). For $\ell_1,\dots,\ell_k \in N$ we define

$$\mathsf{merge}(H,\mathbf{c}(\ell_1,\dots,\ell_k)) := \begin{cases}(H,\ell) & \text{if } H(\ell) = \mathbf{c}(\ell_1,\dots,\ell_k),\\ (H \cup \{\ell_f \mapsto \mathbf{c}(\ell_1,\dots,\ell_k)\},\ell_f) & \text{otherwise.}\end{cases}$$

Observe that the first clause is unambiguous on maximally shared heaps.

Figure 6 collects the small step semantics with respect to a program $\mathtt{P} = (\mathcal{F},\mathcal{C},\mathcal{R})$. We use $\to_{\mathtt{rsm}}$ to abbreviate the relation $\to_{\mathtt{r}} \cup \to_{\mathtt{s}} \cup \to_{\mathtt{m}}$ and likewise we abbreviate $\to_{\mathtt{R}} \cup \to_{\mathtt{rsm}}$ by $\to_{\mathtt{Rrsm}}$. Furthermore, we define $\to_{\mathtt{R/rsm}} := \to_{\mathtt{rsm}}^{*} \cdot \to_{\mathtt{R}} \cdot \to_{\mathtt{rsm}}^{*}$. Hence the *$m$-fold composition* $\to_{\mathtt{R/rsm}}^{m}$ corresponds to a $\to_{\mathtt{Rrsm}}$-reduction with precisely m applications of $\to_{\mathtt{R}}$.

It is now time to show that the model of computation we have just introduced fits our needs, namely that it faithfully simulates big-step semantics as in Figure 5 (itself a correct implementation of call-by-value evaluation from Section 3). This is proved by first showing how big-step semantics can be *simulated* by small-step semantics, later proving that the latter is in fact *deterministic*.

In the following, we denote by $[e]_H$ the term obtained from e by following pointers to the heap, ignoring the annotations $\langle f(\ell_1,\dots,\ell_k),\cdot\rangle$. Formally, we define

$$[e]_H := \begin{cases} f([e_1]_H,\dots,[e_k]_H) & \text{if } e = f(e_1,\dots,e_k),\\ [e']_H & \text{if } e = \langle f(\ell_1,\dots,\ell_k), e'\rangle. \end{cases}$$

Observe that this definition is well-defined as long as H contains all locations occurring in e (a property that is preserved by $\rightarrow_{\mathtt{Rrsm}}$-reductions). An *initial configuration* is a configuration of the form $(\varnothing, H, e)$ with H a maximally shared heap and $e = f(v_1,\dots,v_k)$ an expression unfolding to a function call. Notice that the arguments $v_1,\dots,v_k$ are allowed to contain references to the heap H.

▶ **Lemma 10** (Simulation). *Let $(\varnothing, H, e)$ be an initial configuration. If $(\varnothing, [e]_H) \Downarrow_m (C, v)$ holds for $m \geq 1$ then $(\varnothing, H, e) \rightarrow^m_{\mathtt{R}/\mathtt{rsm}} (D, G, \ell)$ for a location ℓ in G with $[\ell]_G = v$.*

The next lemma shows that the established simulation is *unique*, that is, there is exactly one derivation $(\varnothing, H, e) \rightarrow^m_{\mathtt{R}/\mathtt{rsm}} (D, G, \ell)$. Here, a relation $\rightarrow$ is called *deterministic on a set* A if $b_1 \leftarrow a \rightarrow b_2$ implies $b_1 = b_2$ for all $a \in A$.

▶ **Lemma 11** (Determinism). *The relation $\rightarrow_{\mathtt{Rrsm}}$ is deterministic on all configurations reachable from initial configurations.*

▶ **Theorem 12.** *Suppose $(\varnothing, f(v_1,\dots,v_k)) \Downarrow_m (C, v)$ holds for a reducible term $f(v_1,\dots,v_k)$. Then for each initial configuration $(\varnothing, H, e)$ with $[e]_H = f(v_1,\dots,v_k)$, there exists a unique sequence $(\varnothing, H, e) \rightarrow^m_{\mathtt{R}/\mathtt{rsm}} (D, G, \ell)$ for a location ℓ in G with $[\ell]_G = v$.*

Proof. As $f(v_1,\dots,v_k)$ is reducible, it follows that $m \geq 1$. Hence the theorem follows from Lemma 10 and Lemma 11. ◀

Invariance

Theorem 12 tells us that a term-based semantics (in which sharing is *not* exploited) can be simulated step-by-step by another, more sophisticated, graph-based semantics. The latter's advantage is that each computation step does not require copying, and thus does not increase the size of the underlying configuration too much. This is the key observation towards *invariance*: the number of reduction step is a sensible cost model from a complexity-theoretic perspective. Precisely this will be proved in the remaining of the section.

Define the *size* $|e|$ of an expression recursively by $|\ell| := 1$, $|f(e_1,\dots,e_k)| := 1 + \sum_{i=1}^{k} |e_i|$ and $|\langle f(\ell_1,\dots,\ell_k), e\rangle| := 1 + |e|$. In correspondence we define the *weight* $\mathsf{wt}(e)$ by ignoring locations, i.e. $\mathsf{wt}(\ell) := 0$. Recall that a reduction $(D_1, H_1, e_1) \rightarrow^m_{\mathtt{R}/\mathtt{rsm}} (D_2, H_2, e_2)$ consists of m applications of $\rightarrow_{\mathtt{R}}$, all possibly interleaved by $\rightarrow_{\mathtt{rsm}}$-reductions. As a first step, we thus estimate the overall length of the reduction $(D_1, H_1, e_1) \rightarrow^m_{\mathtt{R}/\mathtt{rsm}} (D_2, H_2, e_2)$ in m and the size of e_1. Set $\Delta := \max\{|r| \mid l \rightarrow r \in \mathcal{R}\}$. The following serves as an intermediate lemma.

▶ **Lemma 13.** *The following properties hold:*
1. *If $(D_1, H_1, e_1) \rightarrow_{\mathtt{rsm}} (D_2, H_2, e_2)$ then $\mathsf{wt}(e_2) < \mathsf{wt}(e_1)$.*
2. *If $(D_1, H_1, e_1) \rightarrow_{\mathtt{R}} (D_2, H_2, e_2)$ then $\mathsf{wt}(e_2) \leq \mathsf{wt}(e_1) + \Delta$.*

Then essentially an application of the *weight gap principle* [18], a form of *amortized* cost analysis, binds the overall length of an $\rightarrow^m_{\mathtt{R}/\mathtt{rsm}}$-reduction suitably.

▶ **Lemma 14.** *If $(D_1, H_1, e_1) \rightarrow^m_{\mathtt{R}/\mathtt{rsm}} (D_2, H_2, e_2)$ then $(D_1, H_1, e_1) \rightarrow^n_{\mathtt{Rrsm}} (D_2, H_2, e_2)$ for $n \leq (1 + \Delta) \cdot m + \mathsf{wt}(e)$ and $\Delta \in \mathbb{N}$ a constant depending only on* $\mathtt{P}$.

Define the size of a configuration $|(D, H, e)|$ as the sum of the sizes of its components. Here, the size $|D|$ of a cache D is defined as its cardinality, similar, the size $|H|$ of a heap is defined as the cardinality of its set of nodes. Notice that a configuration (D, H, e) can be straightforwardly encoded within logarithmic space-overhead as a string $\lceil(D, H, e)\rceil$, i.e. the length of the string $\lceil(D, H, e)\rceil$ is bounded by a function in $O(\log(n) \cdot n)$ in $|(D, H, e)|$, using constants to encode symbols and an encoding of locations logarithmic in $|H|$. Crucially, a step in the small-step semantics increases the size of a configuration only by a constant.

▶ **Lemma 15.** *If $(D_1, H_1, e_1) \to_{\mathtt{Rrsm}} (D_2, H_2, e_2)$ then $|(D_2, H_2, e_2)| \leq |(D_1, H_1, e_1)| + \Delta$.*

▶ **Theorem 16.** *There exists a polynomial $p \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for every initial configuration $(\varnothing, H_1, e_1)$, a configuration (D_2, H_2, e_2) with $(\varnothing, H_1, e_1) \to^{m}_{\mathtt{R/rsm}} (D_2, H_2, e_2)$ is computable from $(\varnothing, H_1, e_1)$ in time $p(|H_1| + |e_1|, m)$.*

Proof. It is tedious, but not difficult to show that the function which implements a step $c \to_{\mathtt{Rrsm}} d$, i.e. which maps $\lceil c\rceil$ to $\lceil d\rceil$, is computable in polynomial time in $\lceil c\rceil$, and thus in the size $|c|$ of the configuration c. Iterating this function at most $n := (1 + \Delta) \cdot m + |(\varnothing, H_1, e_1)|$ times on input $\lceil(\varnothing, H_1, e_1)\rceil$, yields the desired result $\lceil(D_2, H_2, e_2)\rceil$ by Lemma 14. Since each iteration increases the size of a configuration by at most the constant Δ (Lemma 15), in particular the size of each intermediate configuration is bounded by a linear function in $|(\varnothing, H_1, e_1)| = |H_1| + |e_1|$ and n, the theorem follows. ◀

Combining Theorem 12 and Theorem 16 we thus obtain the following.

▶ **Theorem 17.** *There exists a polynomial $p \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for $(\varnothing, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$, the value v represented as DAG is computable from $v_1, \dots, v_k$ in time $p(\sum_{i=1}^{k} |v_i|, m)$.*

Theorem 17 thus confirms that the cost m of a reduction $(\varnothing, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$ is a suitable cost measure. In other words, the *memoized runtime complexity* of a function f, relating input size $n \in \mathbb{N}$ to the maximal cost m of evaluating f on arguments $v_1, \dots, v_k$ of size up to n, i.e. $(\varnothing, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$ with $\sum_{i=1}^{k} |v_i| \leq n$, is an *invariant cost model.*

▶ **Example 18** (Continued from Example 6)**.** Reconsider the program $\mathsf{P_R}$ and the evaluation of a call $\mathit{rabbits}(\mathbf{S}^n(\mathbf{0}))$ which results in the genealogical tree v_n of height $n \in \mathbb{N}$ associated with *Fibonacci's rabbit problem.* Then one can show that $\mathit{rabbits}(\mathbf{S}^n(\mathbf{0})) \Downarrow_m v_n$ with $m \leq 2 \cdot n + 1$. Crucially here, the two intermediate functions a and b defined by simultaneous recursion are called only on proper subterms of the input $\mathbf{S}^n(\mathbf{0})$, hence in particular the rules defining a and b respectively, are unfolded at most n times. As a consequence of the bound on m and Theorem 17 we obtain that the function `rabbits` from the introduction is polytime computable.

▶ Remark. Strictly speaking, our DAG representation of a value v, viz the part of the final heap reachable from a corresponding location ℓ, is not an encoding in the classical, complexity theoretic setting. Different computations resulting in the same value v can produce different DAG representations of v, however, these representations differ only in the naming of locations. Even though our encoding can be exponentially compact in comparison to a linear representation without sharing, it is not exponentially more *succinct* than a reasonable encoding for graphs (e.g. representations as circuits, see Papadimitriou [24]). In such succinct encodings not even equality can be decided in polynomial time. Our form of representation does clearly not fall into this category. In particular, in our setting it can be easily checked in polynomial time that two DAGs represent the same value.

5 GRSR is Sound for Polynomial Time

Sometimes (e.g., in [11]), the first step towards a proof of soundness for ramified recursive systems consists in giving a proper bound precisely relating the size of the result and the size of the inputs. More specifically, if the result has tier n, then the size of it depends polynomially on the size of the inputs of tier higher than n, but only *linearly*, and in very restricted way, on the size of inputs of tier n. Here, a similar result holds, but size is replaced by *minimal shared size*.

The *minimal shared size* $\|v_1, \dots, v_k\|$ for a *sequence* of elements $v_1, \dots, v_k \in \mathbb{A}$ is defined as the number of subterms in $v_1, \dots, v_k$, i.e. the cardinality of the set $\bigcup_{1 \leq i \leq k} \mathsf{STs}(v_i)$. Then $\|v_1, \dots, v_k\|$ corresponds to the number of locations necessary to store the values $v_1, \dots, v_k$ on a heap (compare Lemma 9). If $\mathbf{A}$ is the expression $\mathbb{A}_{n_1} \times \dots \times \mathbb{A}_{n_m}$, n is a natural number, and $\vec{t}$ is a sequence of m terms, then $\|\vec{t}\|^{>n}_{\mathbf{A}}$ is defined to be $\|t_{i_1}, \dots, t_{i_k}\|$ where $i_1, \dots, i_k$ are precisely those indices such that $n_{i_1}, \dots, n_{i_k} > n$. Similarly for $\|\vec{t}\|^{=n}_{\mathbf{A}}$.

▶ **Proposition 19** (Max-Poly). *If* $\mathtt{f} \rhd \mathbf{A} \to \mathbb{A}_n$*, then there is a polynomial* $p_{\mathtt{f}} : \mathbb{N} \to \mathbb{N}$ *such that* $\|\mathtt{f}(\vec{v})\| \leq \|\vec{v}\|^{=n}_{\mathbf{A}} + p_{\mathtt{f}}(\|\vec{v}\|^{>n}_{\mathbf{A}})$.

Once we know that ramified recursive definitions are not too fast-growing for the minimal shared size, we know that all terms around do not have a too-big minimal shared size. As a consequence:

▶ **Proposition 20.** *If* $\mathtt{f} \rhd \mathbf{A} \to \mathbb{A}_n$*, then there is a polynomial* $p_{\mathtt{f}} : \mathbb{N} \to \mathbb{N}$ *such that for every* v*,* $(\varnothing, f(\vec{v})) \Downarrow_m (C, v)$*, with* $m \leq p_{\mathtt{f}}(\|\vec{v}\|)$.

The following, then, is just a corollary of Proposition 20 and Invariance (Theorem 17).

▶ **Theorem 21.** *Let* $\mathtt{f} : \mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_k} \to \mathbb{A}_m$ *be a function defined by general ramified simultaneous recursion. There exists then a polynomial* $p_{\mathtt{f}} : \mathbb{N}^k \to \mathbb{N}$ *such that for all inputs* $v_1, \dots, v_k$*, a DAG representation of* $\mathtt{f}(v_1, \dots, v_k)$ *is computable in time* $p_{\mathtt{f}}(|v_1|, \dots, |v_n|)$.

▶ **Example 22** (Continued from Example 18)**.** In Example 4 we indicated that the function `rabbits` : $\mathbb{N} \to \mathbb{T}$ from Section 2 is definable by GRSR. As a consequence of Theorem 21, it is computable in polynomial time, e.g. on a Turing machine. Similar, we can prove the function `tree` from Section 2 polytime computable.

6 Conclusion

In this work we have shown that simultaneous ramified recurrence on generic algebras is sound for polynomial time, resolving a long-lasting open problem in implicit computational complexity theory. We believe that with this work we have reached the *end of a quest*. Slight extensions, e.g. the inclusion of *parameter substitution*, lead outside polynomial time as soon as simultaneous recursion over trees is permissible.

Towards our main result, we introduced the notion of memoized runtime complexity, and we have shown that this cost model is invariant under polynomial time. Crucially, we use a compact DAG representation of values to control duplication, and tabulation to avoid expensive re-computations. To the authors best knowledge, our work is the first where sharing and memoization are reconciled, in the context of implicit computational complexity theory. Both techniques have been extensively employed, however separately. Essentially relying on sharing, the invariance of the unitary cost model in various rewriting based models of computation, e.g. the λ-calculus [1, 15, 2] and term rewrite systems [14, 5] could be proved.

Several works (e.g. [22, 13, 7]) rely on memoization, employing a measure close to our notion of memoized runtime complexity. None of these works integrate sharing, instead, inputs are either restricted to strings or dedicated bounds on the size of intermediate values have to be imposed. We are confident that our second result is readily applicable to resolve such restrictions.

References

1 B. Accattoli and U. Dal Lago. On the Invariance of the Unitary Cost Model for Head Reduction. In *Proc. of 23rd RTA*, volume 15 of *LIPIcs*, pages 22–37. Dagstuhl, 2012.

2 B. Accattoli and U. Dal Lago. Beta Reduction is Invariant, Indeed. In *Joint Proc. of 23rd CSL and 29th LICS*, page 8. ACM, 2014.

3 T. Arai and N. Eguchi. A New Function Algebra of EXPTIME Functions by Safe Nested Recursion. *TOCL*, 10(4), 2009.

4 M. Avanzini and U. Dal Lago. On Sharing, Memoization, and Polynomial Time. Technical report, University of Bologna, 2014. Available at `http://arxiv.org/abs/1501.00894`.

5 M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 33–48. Dagstuhl, 2010.

6 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

7 P. Baillot, U. Dal Lago, and J.-Y. Moyen. On Quasi-interpretations, Blind Abstractions and Implicit Complexity. *MSCS*, 22(4):549–580, 2012.

8 H. P. Barendregt, M. v. Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *PARLE (2)*, volume 259 of *LNCS*, pages 141–158. Springer, 1987.

9 S. Bellantoni. *Predicative Recursion and Computational Complexity.* PhD thesis, University of Toronto, 1992.

10 S. Bellantoni. Predicative Recursion and the Polytime Hierarchy. In *Feasible Mathematics II.* Birkhäuser Boston, 1994.

11 S. Bellantoni and S. Cook. A new Recursion-Theoretic Characterization of the Polytime Functions. *CC*, 2(2):97–110, 1992.

12 G. Bonfante, R. Kahle, J.-Y. Marion, and I. Oitavem. Recursion Schemata for NCk. In *Proc. of 22nd CSL*, volume 5213 of *LNCS*, pages 49–63. Springer, 2008.

13 G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations: A Way to Control Resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

14 U. Dal Lago and S. Martini. Derivational Complexity is an Invariant Cost Model. In *Revised Selected Papers of 1st FOPARA*, volume 6324 of *LNCS*, pages 100–113. Springer, 2009.

15 U. Dal Lago and S. Martini. On Constructor Rewrite Systems and the Lambda Calculus. *LMCS*, 8(3):1–27, 2012.

16 U. Dal Lago, S. Martini, and M. Zorzi. General Ramified Recurrence is Sound for Polynomial Time. In *Proc. of 1st DICE*, volume 23 of *EPTCS*, pages 47–62, 2010.

17 N. Danner and J. S. Royer. Ramified Structural Recursion and Corecursion. *CoRR*, abs/1201.4567, 2012.

18 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380. Springer, 2008.

19 B. Hoffmann. Term Rewriting with Sharing and Memoization. In *Proc. of 3rd ALP*, volume 632 of *LNCS*, pages 128–142. Springer, 1992.

20 D. Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. of 20th POPL*, pages 325–333. ACM, 1993.

21 D. Leivant. Ramified Recurrence and Computational Complexity I: Word Recurrence and Poly-time. In *Feasible Mathematics II*, volume 13, pages 320–343. Birkhäuser Boston, 1995.
22 J.-Y. Marion. Analysing the Implicit Complexity of Programs. *IC*, 183:2–18, 2003.
23 I. Oitavem. Implicit Characterizations of Pspace. In *PTCS*, pages 170–190, 2001.
24 C. H. Papadimitriou. *Computational Complexity*. Addison Wesley Longman, second edition, 1995.
25 D. Plump. Essentials of Term Graph Rewriting. *ENTCS*, 51:277–289, 2001.