

On Sharing, Memoization, and Polynomial Time[☆]

Martin Avanzini^a, Ugo Dal Lago^a

^a*Università di Bologna & INRIA, Sophia Antipolis*

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.

Keywords: Implicit Computational Complexity, Data-tiering, Polynomial Time

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 *word* algebras, or if recursion is not simultaneous, soundness for polynomial time computation is known to hold [1, 2]. The two soundness results are obtained by quite different means, however: in presence of trees, one is forced to handle *sharing* [2] of common sub-expressions, while simultaneous definitions by recursion requires a form of *memoization* [1].

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:

[☆]This work was partially supported by FWF project number J 3563 and by French ANR project Elica ANR-14-CE25-0005.
Email addresses: martin.avanzini@uibk.ac.at (Martin Avanzini), dallago@cs.unibo.it (Ugo Dal Lago)

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 words, 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 4.15) 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, see also Van Emde Boas [3].
3. We extend upon Leivant’s notion of *ramified recursive functions* [4] 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 5.5). By our invariance theorem, the function algebra is sound for polynomial time, and consequently GRSR characterizes the class of polytime computable functions.

1.1. Related Work

That predicative recursion *on words* is sound for polynomial time, even in presence of simultaneous recursive definitions, is known for a long time [5]. Variations of predicative recursion have been later considered and proved to characterize classes like PH [6], PSPACE [7], EXPTIME [8] or NC [9]. Predicative recursion on trees has been claimed to be sound for polynomial time in the original paper by Leivant [4], the long version of which only deals with words [1]. After fifteen years, the non-simultaneous case has been settled by the second author with Martini and Zorzi [2]; 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 [10]; 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 [11], 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.

This is a revised and extended version of the conference paper [12].

2. The Need for Sharing and Memoization

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.

Let us start with a brief introduction to *ramified recurrence*, for the most simple case where functions are defined over the algebra \mathbb{W} of binary words, build from a constant ϵ , denoting the empty word, and two unary constructors **a** and **b**. We can define the class of *primitive recursive functions*, by closing a small set of *basic functions*, containing the word constructors, projections and a conditional, under composition and *primitive recurrence*. Here, a function **f** is defined by primitive recurrence (or *primitive recursion*) from

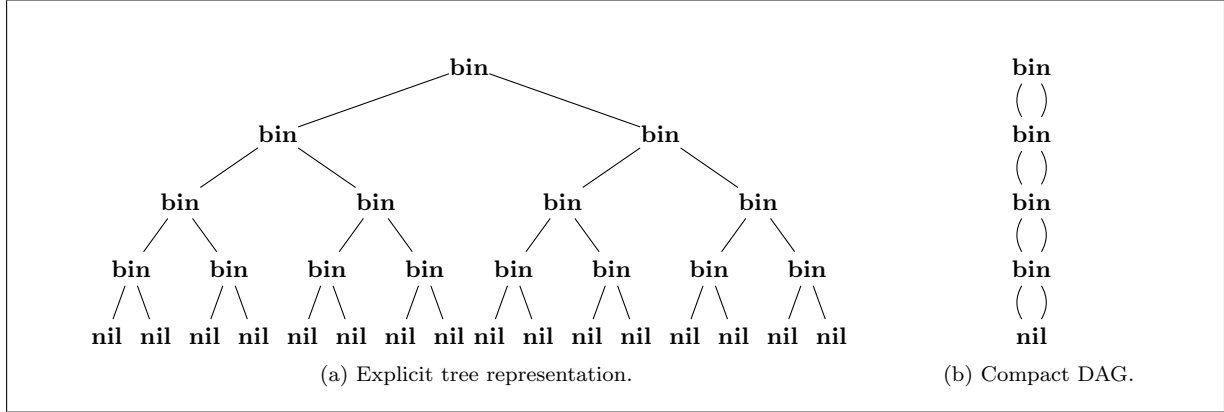


Figure 1: Complete binary tree of height four, as computed by $\mathbf{tree}(\mathbf{S}^4(\mathbf{0}))$.

functions \mathbf{g} , \mathbf{h}_a and \mathbf{h}_b if it is defined by

$$\begin{aligned} \mathbf{f}(\epsilon, \vec{y}) &= \mathbf{g}(\vec{y}) \\ \mathbf{f}(i(x), \vec{y}) &= \mathbf{h}_i(x, \vec{y}, \mathbf{f}(x, \vec{y})) \quad i \in \{\mathbf{a}, \mathbf{b}\}. \end{aligned}$$

The central idea underlying predicative systems, such as the one of Bellantoni and Cook [13] and Leivant [1] for PTIME, is to rule out *impredicative* definitions. In the presence of recursion, impredicativity arises when one of the stepping cases is itself defined by recurrence on the value $\mathbf{f}(x, \vec{y})$, which is about to be defined. In Leivant's system, predicativity is imposed by equipping functions and variables with tiers, tiers then permit one to control the recursion nesting. For each tier $m \in \mathbb{N}$, a copy \mathbb{W}_m of the word algebra \mathbb{W} is singled out. Composition and recursive definitions must preserve tiers. This can be easily expressed as a formal system, in which judgments have the form

$$\mathbf{f} \triangleright \mathbb{W}_{p_1} \times \dots \times \mathbb{W}_{p_k} \rightarrow \mathbb{W}_m,$$

where the naturals p_1, \dots, p_k give the tiers of the inputs, and m the output tier. Thus if \mathbf{f} is defined by primitive recurrence as above, in particular we also have

$$\mathbf{h}_i \triangleright \mathbb{W}_{p_1} \times \dots \times \mathbb{W}_{p_k} \times \mathbb{W}_m \rightarrow \mathbb{W}_m,$$

for the two stepping functions \mathbf{h}_i . The introduction of tiers alone does not limit the expressive power of primitive recurrence. Rather, predicativity is imposed by additionally requiring that the tier p_1 of the recurrence parameter of \mathbf{f} is strictly greater than the output tier m . Hence, any recursion involved in the definition of the stepping function \mathbf{h}_i cannot proceed along the last input, the one of tier m where the recursively computed result is substituted. This form of *ramification* of functions effectively tames primitive recurrence, 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 words to a domain where tree structures are definable, this control is apparently lost. For illustration, let \mathbb{T} denote the algebra of binary trees formed from a constant \mathbf{nil} and a binary constructor \mathbf{bin} . Consider the following definition of a function $\mathbf{tree} : \mathbb{N} \rightarrow \mathbb{T}$, taking a Peano numeral as input.

$$\mathbf{tree}(\mathbf{0}) = \mathbf{nil} \quad \mathbf{tree}(\mathbf{S}(n)) = \mathbf{br}(\mathbf{tree}(n)) \quad \mathbf{br}(t) = \mathbf{bin}(t, t).$$

The function \mathbf{tree} is defined by primitive recurrence, essentially from basic functions. It is easily seen to be ramified in the sense of Leivant, e.g. $\mathbf{tree} \triangleright \mathbb{N}_1 \rightarrow \mathbb{T}_0$ and $\mathbf{br} \triangleright \mathbb{T}_0 \rightarrow \mathbb{T}_0$. Even though the number of recursive steps is linear in the input under a standard reduction order, e.g. call-by-need or call-by-value, the result of $\mathbf{tree}(\mathbf{S}^n(\mathbf{0}))$ 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 \mathbf{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 size n (see Figure 1). Indeed, using the compact DAG representation it is easy to see that the function `tree` is computable in polynomial time. This is the starting point of [2], 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. Here, we allow a sequence of functions $\vec{f} = f_1, \dots, f_k$ to be defined by recurrence in such a way that each function f_i can make use of all the functions \vec{f} in the step case. For the sake of brevity, let us consider the special case where two ternary functions f_1 and f_2 are defined by simultaneous recurrence, over the algebra \mathbb{T} of binary trees. Then the functions f_1, f_2 are defined by *simultaneous recurrence* via the equations

$$\begin{aligned} f_i(\mathbf{nil}, x, y) &= g_i(x, y) ; \\ f_i(\mathbf{bin}(z, w), x, y) &= h_i(z, w, x, y, f_1(z, x, y), f_1(w, x, y), f_2(z, x, y), f_2(w, x, y)) , \end{aligned} \quad (\dagger)$$

for some functions g_1, g_2, h_1 and h_2 . Tiering is naturally extended to the setting of simultaneous recurrence, i.e. $f_1, f_2 \triangleright \mathbb{T}_o \times \mathbb{T}_p \times \mathbb{T}_q \rightarrow \mathbb{T}_m^1$ for natural numbers o, p, q and m with $o > m$, provided

$$g_1, g_2 \triangleright \mathbb{T}_p \times \mathbb{T}_q \rightarrow \mathbb{T}_m \quad \text{and} \quad h_1, h_2 \triangleright \mathbb{T}_o \times \mathbb{T}_o \times \mathbb{T}_p \times \mathbb{T}_q \times \mathbb{T}_m \times \mathbb{T}_m \times \mathbb{T}_m \times \mathbb{T}_m \rightarrow \mathbb{T}_m .$$

For a concrete example, the function which computes the genealogical tree associated with *Fibonacci's rabbit problem* (for $n \in \mathbb{N}$ generations) can be defined via simultaneous ramified recurrence as follows. Let us shortly explain this problem. Rabbits always come in pairs. A *baby* rabbit pair (**B**) matures after one generation, whereas an *adult* rabbit pair (**A**) bears one additional pair of babies at each generation. The problem then asks that starting from a newborn pair of rabbits, how many rabbits will there be after n generations? The following function `rabbits` computes the genealogical tree for a number of generations, given as a Peano number.

$$\begin{aligned} \mathbf{rabbits}(\mathbf{0}) &= \mathbf{B}_L & \mathbf{adult}(\mathbf{0}) &= \mathbf{A}_L & \mathbf{baby}(\mathbf{0}) &= \mathbf{B}_L \\ \mathbf{rabbits}(\mathbf{S}(n)) &= \mathbf{baby}(n) & \mathbf{adult}(\mathbf{S}(n)) &= \mathbf{A}(\mathbf{adult}(n), \mathbf{baby}(n)) & \mathbf{baby}(\mathbf{S}(n)) &= \mathbf{B}(\mathbf{adult}(n)) . \end{aligned}$$

The function `rabbits` is defined by case analysis from two recursive functions `adult` and `baby`. These two function compute the genealogical tree from the perspective of a single adult and baby rabbit pair, respectively. It is not difficult to see that the two functions `adult` and `baby` can be defined by simultaneous recursion, and furthermore, the definition can be easily ramified. Note that in the output of `rabbits(Sn(0))`, which is a tree of depth n , the i^{th} layer corresponds to the rabbit pairs present at the i^{th} generation. Fibonacci's rabbit problem can thus be answered by counting leafs in the output of `rabbits(Sn(0))`. The tree is also 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 over tree structures, indeed, avoiding re-computation of previously computed values becomes more difficult, the trick described above for the case of non-simultaneous definitions 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.

¹ Here and throughout the following, we use the notation $\vec{f} \triangleright \mathbb{T}_{p_1} \times \dots \times \mathbb{T}_{p_k} \rightarrow \mathbb{T}_m$ to denote that $f_i \triangleright \mathbb{T}_{p_1} \times \dots \times \mathbb{T}_{p_k} \rightarrow \mathbb{T}_m$ holds for all functions f_i occurring in the sequence \vec{f} .

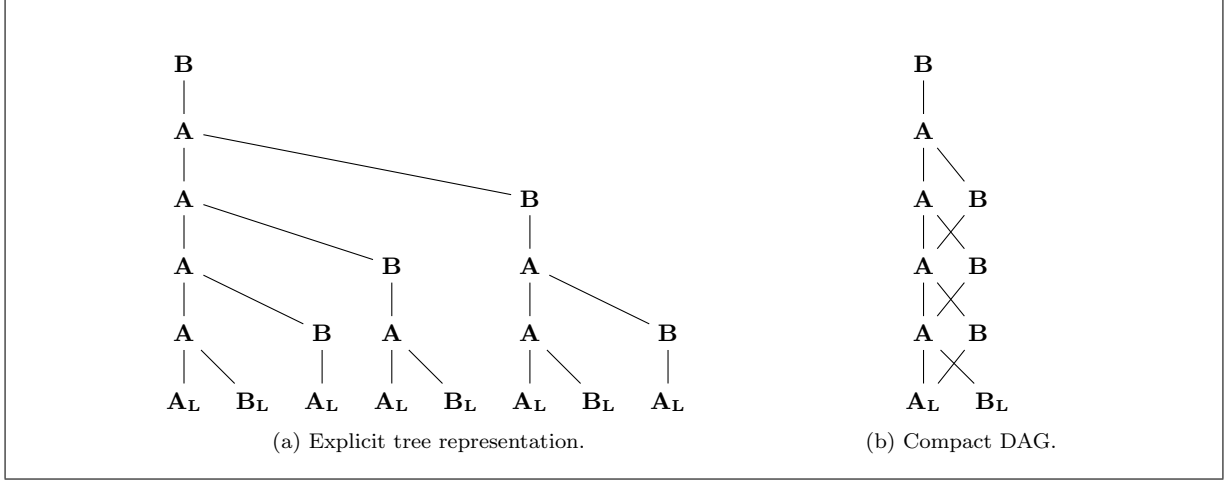


Figure 2: Genealogical rabbit tree up to the sixth generation, as computed by `rabbits(S6(0))`.

3. Preliminaries

3.1. General Ramified Simultaneous Recurrence

Let \mathbb{A} denote a (finite and untyped) signature of constructors $\mathbf{c}_1, \dots, \mathbf{c}_k$, each equipped with an arity $\text{ar}(\mathbf{c}_i) \in \mathbb{N}$. 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} . We are interested in total functions from $\mathbb{A}^n = \underbrace{\mathbb{A} \times \dots \times \mathbb{A}}_{n \text{ times}}$ to \mathbb{A} .

Definition 3.1. The following are so-called *basic functions*:

- For each constructor \mathbf{c} , the *constructor function* $\mathbf{f}_{\mathbf{c}} : \mathbb{A}^{\text{ar}(\mathbf{c})} \rightarrow \mathbb{A}$ for \mathbf{c} , defined as follows:

$$\mathbf{f}_{\mathbf{c}}(x_1, \dots, x_{\text{ar}(\mathbf{c})}) = \mathbf{c}(x_1, \dots, x_{\text{ar}(\mathbf{c})}) .$$

- For each $1 \leq n \leq m$, the (m, n) -*projection function* $\Pi_n^m : \mathbb{A}^m \rightarrow \mathbb{A}$ defined as follows:

$$\Pi_n^m(x_1, \dots, x_m) = x_n .$$

Definition 3.2. We define the class $\text{SIMREC}(\mathbb{A})$ of *simultaneous recursive functions over \mathbb{A}* as the least class of functions that contains the basic functions from Definition 3.1 and that is closed under the following schemes:

- Given a function $\mathbf{f} : \mathbb{A}^n \rightarrow \mathbb{A}$ and n functions $\mathbf{g}_1, \dots, \mathbf{g}_n$, all of them from \mathbb{A}^m to \mathbb{A} , the *composition* $\mathbf{h} = \mathbf{f} \circ (\mathbf{g}_1, \dots, \mathbf{g}_n)$ is a function from \mathbb{A}^m to \mathbb{A} defined as follows:

$$\mathbf{h}(\vec{x}) = \mathbf{f}(\mathbf{g}_1(\vec{x}), \dots, \mathbf{g}_n(\vec{x})) .$$

- Suppose given functions \mathbf{f}_i where $1 \leq i \leq k$ such that for some n , $\mathbf{f}_i : \mathbb{A}^{\text{ar}(\mathbf{c}_i)} \times \mathbb{A}^n \rightarrow \mathbb{A}$. Then the function $\mathbf{g} = \text{case}((\mathbf{f}_i)_{1 \leq i \leq k})$ defined by *case distinction* from $(\mathbf{f}_i)_{1 \leq i \leq k}$ is a function from $\mathbb{A} \times \mathbb{A}^n$ to \mathbb{A} defined as follows:

$$\mathbf{g}(\mathbf{c}_i(\vec{x}), \vec{y}) = \mathbf{f}_i(\vec{x}, \vec{y}) .$$

- Suppose given functions $\mathbf{f}_{i,j}$, where $1 \leq i \leq k$ and $1 \leq j \leq n$, such that for some m , $\mathbf{f}_{i,j} : \mathbb{A}^{\text{ar}(\mathbf{c}_i)} \times \mathbb{A}^m \times \mathbb{A}^{n \cdot \text{ar}(\mathbf{c}_i)} \rightarrow \mathbb{A}$. The functions $(\mathbf{g}_j)_{1 \leq j \leq n} = \text{simrec}((\mathbf{f}_{i,j})_{1 \leq i \leq k, 1 \leq j \leq n})$ defined by *simultaneous primitive recursion* from $(\mathbf{f}_{i,j})_{1 \leq i \leq k, 1 \leq j \leq n}$ are all functions from $\mathbb{A} \times \mathbb{A}^m$ to \mathbb{A} such that for $\vec{x} = x_1, \dots, x_{\text{ar}(\mathbf{c}_i)}$,

$$\mathbf{g}_j(\mathbf{c}_i(\vec{x}), \vec{y}) = \mathbf{f}_{i,j}(\vec{x}, \vec{y}, \mathbf{g}_1(x_1, \vec{y}), \dots, \mathbf{g}_1(x_{\text{ar}(\mathbf{c}_i)}, \vec{y}), \dots, \mathbf{g}_n(x_1, \vec{y}), \dots, \mathbf{g}_n(x_{\text{ar}(\mathbf{c}_i)}, \vec{y})) .$$

We denote by $\text{simrec}((\mathbf{f}_{i,j})_{1 \leq i \leq k, 1 \leq j \leq n})_j$ the j^{th} function \mathbf{g}_j defined by simultaneous primitive recursion.

$$\begin{array}{c}
\frac{}{\mathbf{f}_c \triangleright \mathbb{A}_n^{\text{ar}(c)} \rightarrow \mathbb{A}_n} \quad \frac{}{\Pi_m^n \triangleright \mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_m} \rightarrow \mathbb{A}_{p_n}} \quad \frac{\mathbf{f}_i \triangleright \mathbb{A}_p^{\text{ar}(c_i)} \times \mathbf{A} \rightarrow \mathbb{A}_m}{\text{case}((\mathbf{f}_i)_{1 \leq i \leq k}) \triangleright \mathbb{A}_p \times \mathbf{A} \rightarrow \mathbb{A}_m} \\
\frac{\mathbf{f} \triangleright \mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_n} \rightarrow \mathbb{A}_m \quad \mathbf{g}_i \triangleright \mathbf{A} \rightarrow \mathbb{A}_{p_i}}{\mathbf{f} \circ (\mathbf{g}_1, \dots, \mathbf{g}_n) \triangleright \mathbf{A} \rightarrow \mathbb{A}_m} \quad \frac{\mathbf{f}_{i,j} \triangleright \mathbb{A}_p^{\text{ar}(c_i)} \times \mathbf{A} \times \mathbb{A}_m^{n \cdot \text{ar}(c_i)} \rightarrow \mathbb{A}_m \quad p > m}{\text{simrec}((\mathbf{f}_{i,j})_{1 \leq i \leq k, 1 \leq j \leq n})_l \triangleright \mathbb{A}_p \times \mathbf{A} \rightarrow \mathbb{A}_m}
\end{array}$$

Figure 3: Tiering as a Formal System.

Remark 1. In examples below, we will be slightly more liberal and allow functions whose domain and codomain vary. This could be easily accommodated in the definition of $\text{SIMREC}(\mathbb{A})$, however, at the expense of a more complicated presentation.

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 $\mathbf{f} \triangleright \mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_{\text{ar}(\mathbf{f})}} \rightarrow \mathbb{A}_m$ for $p_1, \dots, p_{\text{ar}(\mathbf{f})}, m$ natural numbers and $\mathbf{f} \in \text{SIMREC}(\mathbb{A})$. The system is defined in Figure 3, where \mathbf{A} denotes the expression $\mathbb{A}_{q_1} \times \dots \times \mathbb{A}_{q_k}$ for some $q_1, \dots, q_k \in \mathbb{N}$. Notice that composition preserves tiers. Moreover, recursion is allowed only on inputs of tier higher than the tier of the function (in the case $\mathbf{f} = \text{simrec}((\mathbf{f}_{i,j})_{1 \leq i \leq k, 1 \leq j \leq n})_l$, we require $p > m$).

Definition 3.3. We call a function $\mathbf{f} \in \text{SIMREC}(\mathbb{A})$ definable by *general ramified simultaneous recurrence* (*GRSR* for short) if $\mathbf{f} \triangleright \mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_{\text{ar}(\mathbf{f})}} \rightarrow \mathbb{A}_m$ holds for some tiers $p_1, \dots, p_{\text{ar}(\mathbf{f})}$ and m .

Remark 2. Consider the *word algebra* \mathbb{W} that was used before, and 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 [1], and consequently all polytime computable functions.

Example 3.4.

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

$$\text{add}(\mathbf{0}, y) = \Pi_1^1(y) = y \quad \text{add}(\mathbf{S}(x), y) = (\mathbf{f}_S \circ \Pi_3^3)(x, y, \text{add}(x, y)) = \mathbf{S}(\text{add}(x, y)),$$

using general simultaneous ramified recursion, i.e. $\text{add} = \text{simrec}((\Pi_1^1, \mathbf{f}_S \circ \Pi_3^3))_1$.

2. Let $\mathbb{F} := \{\mathbf{B}_L, \mathbf{A}_L, \mathbf{B}, \mathbf{A}\}$, where $\text{ar}(\mathbf{B}_L) = \text{ar}(\mathbf{A}_L) = 0$, $\text{ar}(\mathbf{B}) = 1$ and $\text{ar}(\mathbf{A}) = 2$. Then we can define the functions $\text{rabbits} \triangleright \mathbb{N}_i \rightarrow \mathbb{F}_j$ for all i and j with $i > j$ from Section 2 by case analysis from the functions adult and baby , defined by simultaneous ramified recurrence as follows.

$$\begin{aligned} \text{adult}(\mathbf{0}) &= \mathbf{f}_{\mathbf{A}_L} = \mathbf{A}_L & \text{adult}(\mathbf{S}(n)) &= (\mathbf{f}_A \circ (\Pi_2^3, \Pi_3^3))(n, \text{adult}(n), \text{baby}(n)) = \mathbf{A}(\text{adult}(n), \text{baby}(n)) \\ \text{baby}(\mathbf{0}) &= \mathbf{f}_{\mathbf{B}_L} = \mathbf{B}_L & \text{baby}(\mathbf{S}(n)) &= (\mathbf{f}_B \circ \Pi_3^3)(n, \text{adult}(n), \text{baby}(n)) = \mathbf{B}(\text{adult}(n)). \end{aligned}$$

3. We can define a function $\#\text{leafs}$ from \mathbb{F} to \mathbb{N} by simultaneous primitive recursion, which counts the number of leafs in trees as follows.

$$\begin{aligned} \#\text{leafs}(\mathbf{B}_L) &= \mathbf{S}(\mathbf{0}) & \#\text{leafs}(\mathbf{A}_L) &= \mathbf{S}(\mathbf{0}) \\ \#\text{leafs}(\mathbf{B}(t)) &= \#\text{leafs}(t) & \#\text{leafs}(\mathbf{A}(l, r)) &= \text{add}(\#\text{leafs}(l), \#\text{leafs}(r)). \end{aligned}$$

However, this function *cannot* be ramified, since add in the last equation requires different tiers. Indeed, having a ramified recursive function $\#\text{leafs} \triangleright \mathbb{F}_i \rightarrow \mathbb{N}_0$ (for some $i > 0$) defined as above would allow us to ramify $\text{fib} = \#\text{leafs} \circ \text{rabbits}$ which on input n computes the n^{th} Fibonacci number, and is thus an exponential function.

$$\begin{array}{c}
\frac{t_i \downarrow v_i \quad f(p_1, \dots, p_k) \rightarrow r \in \mathcal{R} \quad p_i \sigma = v_i \quad r \sigma \downarrow v}{f(t_1, \dots, t_k) \downarrow v} \text{ (operator)} \\
\\
\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \dots, t_k) \downarrow \mathbf{c}(v_1, \dots, v_k)} \text{ (constructor)}
\end{array}$$

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

3.2. Computational Model, Syntax and Semantics

We assume modest familiarity with rewriting, see the book of Baader and Nipkow [14] for a gentle introduction. We now 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* $\text{STs}(t)$ of a term t is defined by $\text{STs}(t) := \{t\}$ if $t \in \mathcal{V}$ and $\text{STs}(t) := \bigcup_{1 \leq i \leq \text{ar}(f)} \text{STs}(t_i) \cup \{t\}$ if $t = f(t_1, \dots, t_{\text{ar}(f)})$. A *position* is a finite sequence of positive natural numbers. We denote by ε the empty position, and by $p \cdot q$ the concatenation of positions p and q . The subterm $t|_p \in \text{STs}(t)$ of t at position p is defined as follows: $t|_\varepsilon := t$ and if $t = f(t_1, \dots, t_{\text{ar}(f)})$ and $1 \leq i \leq \text{ar}(f)$ then $t|_{i \cdot p} = t_i|_p$. We say that position p is a position in t if $t|_p$ is well-defined. 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 the domain of σ by $\sigma(x)$.

Definition 3.5. A *program* 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* $l \rightarrow r$ over terms $l, r \in \mathcal{T}(\mathcal{F} \cup \mathcal{C}, \mathcal{V})$, satisfying:

- the *left-hand side* l is of the form $f(p_1, \dots, p_k)$ with $f \in \mathcal{F}$ and all arguments p_j are *patterns*, i.e. terms formed from the constructors of \mathcal{C} and variables; and
- all variables occurring in the *right-hand side* r also occur in the left-hand side l .

Furthermore, we require that the set of rules \mathcal{R} is *orthogonal*, that is, the following two requirements are met:

- *left-linearity*: the left-hand side l of each rule $l \rightarrow r \in \mathcal{R}$ is *linear*; and
- *non-ambiguity*: there are no two rules with overlapping left-hand sides in \mathcal{R} .

We keep the program $P = (\mathcal{F}, \mathcal{C}, \mathcal{R})$ fixed throughout the following. Our notion of programs corresponds to the one of *orthogonal constructor rewrite systems*, which define a class of *deterministic* first-order functional programs, see e.g. [14]. 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 below.

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 P computes the function $\mathbf{f} : \mathcal{T}(\mathcal{C})^k \rightarrow \mathcal{T}(\mathcal{C})$ if there exists an operation symbol $f \in \mathcal{F}$ such that $\mathbf{f}(v_1, \dots, v_k) = v$ if and only if $f(v_1, \dots, v_k) \downarrow v$ holds for all inputs $v_i \in \mathcal{T}(\mathcal{C})$.

Example 3.6 (Continued from Example 3.4). The definition of **rabbits** from Section 2 can be turned into a program P_{rabbits} , by *orienting* the underlying equations from left to right, replacing applications of functions \mathbf{f} with corresponding operation symbols f . More precisely:

- the set of operation symbols underlying P_{rabbits} consists of (i) ternary symbols P_2^3 and P_3^3 implementing the projections Π_2^3 and Π_3^3 ; (ii) constant operation symbols $f_{\mathbf{A}_L}, f_{\mathbf{B}_L}$, a binary operation symbol $f_{\mathbf{A}}$ and a unary operation symbol $f_{\mathbf{B}}$, all implementing the constructor functions associated with \mathbb{F} ; (iii) two

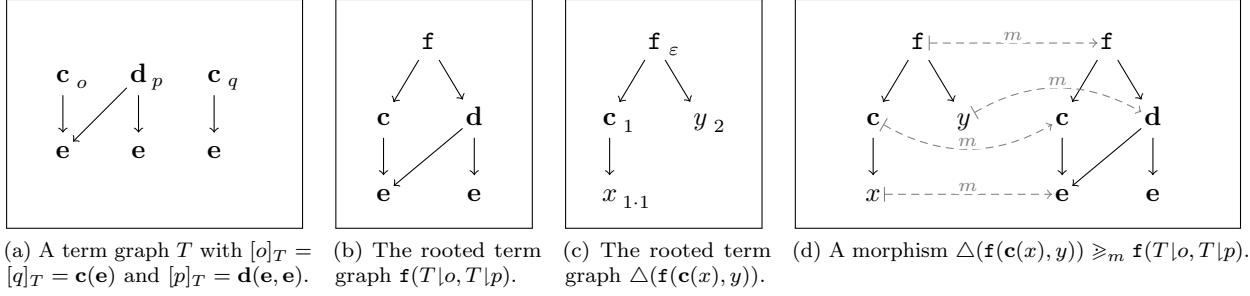


Figure 5: Term graphs and morphisms.

operation symbols $\mathit{comp}[f_{\mathbf{A}}; P_2^3, P_3^3]$ and $\mathit{comp}[f_{\mathbf{B}}, P_3^3](x_1, x_2, x_3)$ implementing the composition of the mentioned operations; (iv) two operation symbols

$$\begin{aligned} a &:= \mathit{simrec}[f_{\mathbf{A}_L}, \mathit{comp}[f_{\mathbf{A}}, P_2^3, P_3^3], f_{\mathbf{B}_L}, \mathit{comp}[f_{\mathbf{B}}, P_3^3]]_1; \\ b &:= \mathit{simrec}[f_{\mathbf{A}_L}, \mathit{comp}[f_{\mathbf{A}}, P_2^3, P_3^3], f_{\mathbf{B}_L}, \mathit{comp}[f_{\mathbf{B}}, P_3^3]]_2, \end{aligned}$$

implementing the recursively defined functions `adult` and `baby`; and (v) an operation symbol $\mathbf{r} := \mathit{case}[f_{\mathbf{B}_L}, b]$ implementing the function `rabbits`;

- the set of constructors underlying $\mathbf{P}_{\mathit{rabbits}}$ consists of the constructors from $\mathbb{F} \cup \mathbb{N}$;
- the set of rules underlying $\mathbf{P}_{\mathit{rabbits}}$ is given as follows, where $\vec{x} := x_1, x_2, x_3$.

$$\begin{array}{lll} P_2^3(\vec{x}) \rightarrow x_2 & f_{\mathbf{A}_L} \rightarrow \mathbf{A}_L & f_{\mathbf{A}}(x_1, x_2) \rightarrow \mathbf{A}(x_1, x_2) \\ P_3^3(\vec{x}) \rightarrow x_3 & f_{\mathbf{B}_L} \rightarrow \mathbf{B}_L & f_{\mathbf{B}}(x_1) \rightarrow \mathbf{B}_L(x_1) \\ a(\mathbf{0}) \rightarrow f_{\mathbf{A}_L} & a(\mathbf{S}(n)) \rightarrow \mathit{comp}[f_{\mathbf{A}}, P_2^3, P_3^3](n, a(n), b(n)) & \mathit{comp}[f_{\mathbf{A}}, P_2^3, P_3^3](\vec{x}) \rightarrow f_{\mathbf{A}}(P_2^3(\vec{x}), P_3^3(\vec{x})) \\ b(\mathbf{0}) \rightarrow f_{\mathbf{B}_L} & b(\mathbf{S}(n)) \rightarrow \mathit{comp}[f_{\mathbf{B}}, P_3^3](n, a(n), b(n)) & \mathit{comp}[f_{\mathbf{B}}, P_3^3](\vec{x}) \rightarrow f_{\mathbf{B}}(P_3^3(\vec{x})) \\ \mathbf{r}(\mathbf{0}) \rightarrow f_{\mathbf{B}_L} & \mathbf{r}(\mathbf{S}(n)) \rightarrow b(n). \end{array}$$

It is not difficult to see that by construction, $\mathbf{P}_{\mathit{rabbits}}$ computes the function `rabbits` from Section 2.

The construction is straight-forward to generalize to arbitrary functions in $\mathbf{SIMREC}(\mathbb{A})$, see e.g. the work on term rewriting characterizations by Cichon and Weiermann [15] for a more formal treatment.

Definition 3.7. Let $\mathbf{f} \in \mathbf{SIMREC}(\mathbb{A})$. We define the *program* $\mathbf{P}_{\mathbf{f}}$ associated with \mathbf{f} as $(\mathcal{F}_{\mathbf{f}}, \mathcal{C}_{\mathbf{f}}, \mathcal{R}_{\mathbf{f}})$, where:

- the set of operations $\mathcal{F}_{\mathbf{f}}$ contains for each function \mathbf{g} underlying the definition of \mathbf{f} a fresh operation symbol \mathbf{g} ;
- the set of constructors $\mathcal{C}_{\mathbf{f}}$ contains the constructors of \mathbb{A} ;
- the set of rules $\mathcal{R}_{\mathbf{f}}$ contains for each equation $l = r$ underlying the definition of a function \mathbf{g} associated with $\mathbf{g} \in \mathcal{F}_{\mathbf{f}}$ the *orientation* $\langle l \rangle \rightarrow \langle r \rangle$, where the terms $\langle l \rangle$ and $\langle r \rangle$ are obtained by replacing applications of functions with the corresponding operation symbols in l and r , respectively.

Notice that due to the inductive definition of the class $\mathbf{SIMREC}(\mathbb{A})$, the program $\mathbf{P}_{\mathbf{f}}$ is finite.

Proposition 3.8. For each $\mathbf{f} \in \mathbf{SIMREC}(\mathbb{A})$, the program $\mathbf{P}_{\mathbf{f}}$ associated with \mathbf{f} computes the function \mathbf{f} .

3.3. Term Graphs

We borrow key concepts from *term graph rewriting* (see e.g. the survey of Plump [16] for an overview) and follow the presentation of Barendregt et al. [17]. 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 $(N, \text{suc}, \text{lab})$ consisting of *nodes* N , a *successors function* $\text{suc} : N \rightarrow N^*$ and a *labeling function* $\text{lab} : N \rightarrow \mathcal{F} \cup \mathcal{V}$. We require that term graphs are *compatible* with \mathcal{F} , in the sense that for each node $o \in N$, if $\text{lab}(o) = f \in \mathcal{F}$ then $\text{suc}(o) = [o_1, \dots, o_{\text{ar}(f)}]$ and otherwise, if $\text{lab}(o) = x \in \mathcal{V}$, $\text{suc}(o) = []$. In the former case, we also write $T(o) = f(o_1, \dots, o_{\text{ar}(f)})$, the latter case is denoted by $T(o) = x$. We define the *successor relation* \rightarrow_T on nodes in T such that $o \rightarrow_T p$ holds iff p occurs in $\text{suc}(o)$, if p occurs at the i^{th} position we also write $o \xrightarrow{i}_T p$. Throughout the following, we consider only *acyclic* term graphs, that is, when \rightarrow_T is acyclic. Hence the *unfolding* $[o]_T$ of T at node o , defined as follows, results in a finite term.

$$[o]_T := \begin{cases} x & \text{if } T(o) = x \text{ is a variable,} \\ f([o_1]_T, \dots, [o_k]_T) & \text{if } T(o) = f(o_1, \dots, o_k). \end{cases}$$

We called the term graph T *rooted* if there exists a unique node o , the *root* of T , with $o \rightarrow_T^* p$ for every $p \in N$. We denote by $T|_o$ the *sub-graph* of T rooted at o . For a symbol $f \in \mathcal{F}$ and nodes $\{o_1, \dots, o_{\text{ar}(f)}\} \subseteq N$, we write $f(T|_{o_1}, \dots, T|_{o_{\text{ar}(f)}})$ for the term graph $S|_{o_f}$, where S is defined as the extension of T by a fresh node $o_f \notin N$ with $S(o_f) = f(o_1, \dots, o_{\text{ar}(f)})$.

Every (linear) term t is representable as a *canonical tree* $\Delta(t) = (N, \text{suc}, \text{lab})$ unfolding to t as follows. The nodes N are precisely the positions in t . The successors function is defined by $\text{suc}(p) := []$ if $t|_p$ is a variable, otherwise $t|_p = f(t_1, \dots, t_{\text{ar}(f)})$ and $\text{suc}(p) := [p \cdot 1, \dots, p \cdot \text{ar}(f)]$. Compare Figure 5a, b and, c, which illustrate key concepts.

For two rooted term graphs $T = (N_T, \text{suc}_T, \text{lab}_T)$ and $S = (N_S, \text{suc}_S, \text{lab}_S)$, a mapping $m : N_T \rightarrow N_S$ is called *morphic* in $o \in N_T$ if (i) $\text{lab}_T(o) = \text{lab}_S(m(o))$ and (ii) $o \xrightarrow{i}_T p$ implies $m(o) \xrightarrow{i}_S m(p)$ for all appropriate i . A *homomorphism* from T to S is a mapping $m : N_T \rightarrow N_S$ 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 $T \geq_m S$ to indicate that m is a homomorphism from T to S . See also Figure 5d for an example.

The following proposition relates matching on terms and homomorphisms on graphs. The proposition essentially relies on the imposed linearity condition. Note that for a linear term t , to each variable x in t we can associate a *unique node* in $\Delta(t)$ labeled by x , which we denote by p_x below.

Proposition 3.9 (Matching on Graphs). *Let t be a linear term, let T be a term graph and let o be a node of T .*

1. *If $\Delta(t) \geq_m T|_o$ 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 $\Delta(t) \geq_m T|_o$.*

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

Proof. The proof of both properties is by structural induction on t . We first prove the direction (1). Assume $\Delta(t) \geq_m T|_o$. When t is a variable, the substitution $\sigma := \{t \mapsto [o]_T\}$ satisfies $t\sigma = [o]_T$. Since $m(\varepsilon) = o$ by definition, we conclude the base case. For the inductive step, assume $t = f(t_1, \dots, t_k)$. The assumption $\Delta(t) \geq_m T|_o$ yields $T(o) = f(o_1, \dots, o_k)$ for some nodes o_i in T . For $i = 1, \dots, k$, define m_i by $m_i(p) := m(i \cdot p)$ for each position $i \cdot p$ in t . By case analysis on the nodes of $\Delta(t_i)$ one verifies $\Delta(t_i) \geq_{m_i} [o_i]_T$. Thus the induction hypothesis (IH) yields substitutions σ_i for $i = 1, \dots, k$, such that $t_i\sigma_i = [o_i]_T$. Without loss of generality, suppose σ_i is restricted to variables in t_i . By linearity of t , the domains of the substitutions σ_i are pairwise disjoint. Thus the union $\sigma := \bigcup_{i=1}^k \sigma_i$ gives a well-defined substitution. As we have $t\sigma = f(t_1\sigma_1, \dots, t_k\sigma_k) = f([o_1]_T, \dots, [o_k]_T) = [o]_T$, the inductive step follows.

For the direction (2), suppose $t\sigma = [o]_T$. If t is a variable and thus $\Delta(t)$ consists of a single node labeled by t , trivially $\Delta(t) \geq_m T|_o$ holds for m the homomorphism which maps the root node ε of $\Delta(t)$ to o . For the inductive step suppose $t = f(t_1, \dots, t_k)$, hence $T(o) = f(o_1, \dots, o_k)$, $t_i\sigma = [o_i]_T$ ($i = 1, \dots, k$) and thus

$$\begin{array}{c}
\frac{f \in \mathcal{F} \quad (C_{i-1}, t_i) \Downarrow_{n_i} (C_i, v_i) \quad (C_k, f(v_1, \dots, v_k)) \Downarrow_n (C_{k+1}, v) \quad m = n + \sum_{i=1}^k n_i}{(C_0, f(t_1, \dots, t_k)) \Downarrow_m (C_{k+1}, v)} \text{(\mathcal{F-context})} \\
\\
\frac{\mathbf{c} \in \mathcal{C} \quad (C_{i-1}, t_i) \Downarrow_{n_i} (C_i, v_i) \quad m = \sum_{i=1}^k n_i}{(C_0, \mathbf{c}(t_1, \dots, t_k)) \Downarrow_m (C_k, \mathbf{c}(v_1, \dots, v_k))} \text{(\mathcal{C-context})} \\
\\
\frac{(f(v_1, \dots, v_k), v) \in \mathcal{C}}{(C, f(v_1, \dots, v_k)) \Downarrow_0 (C, v)} \text{(read)} \\
\\
\frac{(f(v_1, \dots, v_k), v) \notin \mathcal{C} \quad f(p_1, \dots, p_k) \rightarrow r \in \mathcal{R} \quad \forall i. p_i \sigma = v_i \quad (C, r\sigma) \Downarrow_m (D, v)}{(C, f(v_1, \dots, v_k)) \Downarrow_{m+1} (D \cup \{(f(v_1, \dots, v_k), v)\}, v)} \text{(update)}
\end{array}$$

Figure 6: Cost annotated operational semantics with memoization for program $(\mathcal{F}, \mathcal{C}, \mathcal{R})$.

by IH $\Delta(t_i) \cong_{m_i} [o_i]_T$ for homomorphisms m_i . Define the function m by $m(\varepsilon) := o$ and $m(i.p) := m_i(p)$ for all $i = 1, \dots, k$ and positions $i.p$ of t . Observe that m is defined on all nodes of $\Delta(t)$. By definition of m one finally concludes the lemma, using the IH together with the equalities $\Delta(t)(i.p) = \Delta(t_i)(p)$ for positions $i.p$ ($i = 1, \dots, k$) of $\Delta(t)$. \square

4. Memoization and Sharing, Formally

In the following, we first extend our simple computational model with *memoization*. Memoization is applied pervasively. During evaluation, the result of *each* function call is tabulated in a *cache*, i.e. a lookup table, repetitive function calls are replaced by lookups to the cache. The extension with memoization, expressed in an *operational semantics* style, is standard, see e.g. [18, 19]. These semantics give rise to a *natural, unitary cost model* for programs, the *memoized runtime complexity*. Essentially, the memoized runtime complexity of a program accounts for the number of evaluation steps, but memoized calls are free.

In the second part of this section, we then show that this cost model does not underestimate the complexity of programs too much. To this end, in Section 4.1 we provide a *small-step semantics*, itself an implementation of the operational semantics, that also takes sharing into account. This in turn, allows us to prove in Section 4.2 that the memoized runtime complexity is a *reasonable* cost model for programs, i.e. each program admits an implementation such that the runtime of this implementation can be bounded by a polynomial in the memoized runtime complexity.

Throughout the following, we fix a program $P = (\mathcal{F}, \mathcal{C}, \mathcal{R})$. To integrate memoization, we make use of a *cache* C to store results of intermediate functions calls. Caches are modeled as a sets of tuples $(f(v_1, \dots, v_{\text{ar}(f)}), u)$, with $f \in \mathcal{F}$ and $v_1, \dots, v_{\text{ar}(f)}$ as well as u are values. We require that caches C are *proper* with respect to the program P , i.e. C associates function calls $f(v_1, \dots, v_k)$ with their corresponding result u . Formally, a cache C is called proper if $(f(v_1, \dots, v_k), u) \in C$ implies $f(v_1, \dots, v_k) \Downarrow u$.

Figure 6 collects the *memoizing operational semantics* of the program P . 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 rule **(operator)** from Figure 4 is split into the three rules **(\mathcal{F} -context)**, **(read)** and **(update)**. With rule **(\mathcal{F} -context)** the arguments are first evaluated to values. Once evaluated, depending on whether the corresponding function call is already *tabulated*, either with rule **(read)** the result is loaded from the cache, or with rule **(update)** the reduction is performed and the cache accordingly updated. 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. The following lemma confirms that the call-by-value semantics of Section 3 is correctly implemented by the memoizing semantics.

Lemma 4.1. *We have $(C, t) \Downarrow_m (D, v)$ for some $m \in \mathbb{N}$ and proper cache C if and only if $t \Downarrow v$.*

Proof. We consider the direction from right to left first. The proof is by induction on the deduction Π of the statement $t \Downarrow v$, showing additionally that the cache D is proper.

- Suppose that the last rule in Π has the form

$$\frac{t_i \Downarrow v_i \quad \mathbf{f}(p_1, \dots, p_k) \rightarrow r \in \mathcal{R} \quad p_i \sigma = v_i \quad r \sigma \Downarrow v}{\mathbf{f}(t_1, \dots, t_k) \Downarrow v} \text{ (operator) } .$$

By IH, we obtain proper caches D_0, \dots, D_k with $D_0 = C$ and $(D_{i-1}, t_i) \Downarrow_{m_i} (D_i, v_i)$. By the rule (\mathcal{F} -context), it suffices to show $(D_k, \mathbf{f}(v_1, \dots, v_k)) \Downarrow_n (D, v)$ for D a proper cache. We distinguish two cases. Consider the case $(\mathbf{f}(v_1, \dots, v_k), u) \in D_k$ for some u . Since D_k is proper, we have $\mathbf{f}(v_1, \dots, v_k) \Downarrow u$. Since the program \mathcal{R} is orthogonal and thus confluent, it is not difficult to see that $v = u$. We conclude the case by one application of rule (**read**). Otherwise, when $\mathbf{f}(v_1, \dots, v_k)$ is not cached, we conclude by rule (**update**) using the IH on $r \sigma \Downarrow v$. Note that the resulting cache is also in this case proper.

- The second and final case, where the last rule applied in Π is rule (**constructor**), follows directly from IH and rule (\mathcal{C} -context).

For the direction from left to right we proceed by induction on the deduction Π of $(C, t) \Downarrow_m (D, v)$.

- Suppose first that the last rule in Π is of the form:

$$\frac{(C_{i-1}, t_i) \Downarrow_{m_i} (C_i, v_i) \quad (C_k, \mathbf{f}(v_1, \dots, v_k)) \Downarrow_n (C_{k+1}, v)}{(C_0, \mathbf{f}(t_1, \dots, t_k)) \Downarrow_m (C_{k+1}, v)} \text{ (}\mathcal{F}\text{-context) } .$$

Observe that as in the previous direction, all involved caches are proper. Thus by IH, we see that $t_i \Downarrow v_i$ and $\mathbf{f}(v_1, \dots, v_k) \Downarrow v$ holds, where the latter statement is due to rule (**operator**). Collecting all the pre-conditions, it is not difficult to derive $\mathbf{f}(t_1, \dots, t_k) \Downarrow v$ by one application of rule (**operator**).

- The remaining cases, where the last rule in Π is (\mathcal{C} -context), (**read**) or (**update**), follow either from the assumption that C is proper, or directly from IH. \square

The revised operational semantics account for memoization, but do not overcome the size explosion problem observed in Section 2. To tame growth rates in value sizes, we now define *small-step semantics* corresponding to the memoizing semantics, facilitating sharing of common sub-expressions.

4.1. Small-Step Semantics with Memoization and Sharing

To incorporate sharing, we extend the pair (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* v , and define *expressions* e and (*evaluation*) *contexts* E according to the following grammar:

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

Here, $\ell_1, \dots, \ell_k, \ell \in \text{Loc}$, $\mathbf{f} \in \mathcal{F}$ 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 $\langle \mathbf{f}(\ell_1, \dots, \ell_k), e \rangle$ indicates that the partially evaluated expression e descends from a call $\mathbf{f}(v_1, \dots, v_k)$, with arguments v_i stored at location ℓ_i on the heap. A context E is an expression with a unique *hole*, denoted as \square , 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 , a *heap* H and an expression e . Unlike before, the cache D consists of pairs of the form $(\mathbf{f}(\ell_1, \dots, \ell_k), \ell)$ where, instead of values, we store references

$$\begin{array}{c}
\frac{(f(\ell_1, \dots, \ell_k), \ell) \notin D \quad f(p_1, \dots, p_k) \rightarrow r \in \mathcal{R} \quad \Delta(f(p_1, \dots, p_k)) \gg_m f(H|\ell_1, \dots, H|\ell_k) \quad \sigma_m := \{x \mapsto m(\ell_x) \mid \ell_x \in \text{Loc}, T(\ell_x) = x \in \mathcal{V}\}}{(D, H, E[f(\ell_1, \dots, \ell_k)]) \rightarrow_{\mathbf{R}} (D, H, E[\langle f(\ell_1, \dots, \ell_k), r\sigma_m \rangle])} \text{ (apply)} \\
\\
\frac{(f(\ell_1, \dots, \ell_k), \ell) \in D}{(D, H, E[f(\ell_1, \dots, \ell_k)]) \rightarrow_{\mathbf{r}} (D, H, E[\ell])} \text{ (read)} \\
\\
\frac{}{(D, H, E[\langle f(\ell_1, \dots, \ell_k), \ell \rangle]) \rightarrow_{\mathbf{s}} (D \cup \{(f(\ell_1, \dots, \ell_k), \ell)\}, H, E[\ell])} \text{ (store)} \\
\\
\frac{(H', \ell) = \text{merge}(H, \mathbf{c}(\ell_1, \dots, \ell_k))}{(D, H, E[\mathbf{c}(\ell_1, \dots, \ell_k)]) \rightarrow_{\mathbf{m}} (D, H', E[\ell])} \text{ (merge)}
\end{array}$$

Figure 7: Small-step semantics with memoization and sharing for program $(\mathcal{F}, \mathcal{C}, \mathcal{R})$.

$\ell_1, \dots, \ell_k, \ell$ 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. By definition thus, heaps are always acyclic. 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 any two locations ℓ_a, ℓ_b of H . The operation $\text{merge}(H, \mathbf{c}(\ell_1, \dots, \ell_k))$, defined as follows, is used to extend the heap H with a constructor \mathbf{c} whose arguments point to ℓ_1, \dots, ℓ_k , retaining maximal sharing. For $\ell_1, \dots, \ell_k \in N$ we define

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

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

Figure 7 defines small-step semantics for the program P as the composition of four relations $\rightarrow_{\mathbf{R}}, \rightarrow_{\mathbf{r}}, \rightarrow_{\mathbf{s}}$ and $\rightarrow_{\mathbf{m}}$, which are defined by the rules **(apply)**, **(read)**, **(store)** and **(merge)**, respectively. The relation $\rightarrow_{\mathbf{R}}$, defining rule application, is used to implement the rule **(\mathcal{F} -context)** from the memoizing operational semantics. Matching is performed in accordance to Proposition 3.9, and variables are instantiated by the corresponding locations of values on the heap. Once the reduct is evaluated to a value stored at a location ℓ on the heap, the relation $\rightarrow_{\mathbf{s}}$ removes this marker and adds a corresponding entry to the heap. Finally, the relation $\rightarrow_{\mathbf{r}}$ implements memoization as in the operational semantics, the relation $\rightarrow_{\mathbf{m}}$ ensures that values are internalized in the heap.

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 6 (itself a correct implementation of call-by-value evaluation from Section 3). This is proven by first showing how big-step semantics can be *simulated* by small-step semantics, later proving that the latter is in fact *deterministic*. Throughout the following, we abbreviate with \rightarrow_{rsm} the relation $\rightarrow_{\mathbf{r}} \cup \rightarrow_{\mathbf{s}} \cup \rightarrow_{\mathbf{m}}$, likewise we abbreviate $\rightarrow_{\mathbf{R}} \cup \rightarrow_{\text{rsm}}$ by $\rightarrow_{\text{Rrsm}}$. Furthermore, we define $\rightarrow_{\text{R/rsm}} := \rightarrow_{\text{rsm}}^* \cdot \rightarrow_{\mathbf{R}} \cdot \rightarrow_{\text{rsm}}^*$. Hence the *m-fold composition* $\rightarrow_{\text{R/rsm}}^m$ corresponds to a $\rightarrow_{\text{Rrsm}}$ -reduction with precisely m applications of $\rightarrow_{\mathbf{R}}$. We are interested in reductions over *well-formed* configurations.

Definition 4.2. A configuration (D, H, e) is *well-formed* if the following conditions hold.

1. The heap H is maximally shared.
2. The cache D is a function and *compatible* with e . Here, compatibility means that if $\langle f(\ell_1, \dots, \ell_k), e' \rangle$ occurs as a sub-expression in e , then $(f(\ell_1, \dots, \ell_k), \ell) \notin D$ for any ℓ .
3. The configuration contains no *dangling references*, i.e. $H(\ell)$ is defined for each location ℓ occurring in D and e .

The following lemma confirms that well-formed configurations are preserved by reductions.

Lemma 4.3.

1. If $(D, H, E[e])$ is well-formed then so is (D, H, e) .
2. If $(D_1, H_1, e_1) \rightarrow_{\text{Rrsm}} (D_2, H_2, e_2)$ and (D_1, H_1, e_1) is well-formed then so is (D_2, H_2, e_2) .

Proof. It is not difficult to see that Assertion 1 holds. To see that Assertion 2 holds, fix a well-formed configuration (D_1, H_1, e_1) and suppose $(D_1, H_1, e_1) \rightarrow_{\text{Rrsm}} (D_2, H_2, e_2)$. We check that (D_2, H_2, e_2) is well-formed by case analysis on $\rightarrow_{\text{Rrsm}}$.

1. *The heap H_2 is maximally shared:* As only the relation \rightarrow_{m} modifies the heap, it suffices to consider the case $(D_1, H_1, e_1) \rightarrow_{\text{m}} (D_2, H_2, e_2)$. Then $(H_2, \ell) = \text{merge}(H_1, \mathbf{c}(\ell_1, \dots, \ell_k))$ for some location ℓ , and the property follows as merge preserves maximal sharing.
2. *The cache D_2 is a function and compatible with e_2 :* Only the relation \rightarrow_{s} updates the cache. By compatibility of D_1 with e_1 it follows then that D_2 is a function. Concerning compatibility, only the rules \rightarrow_{r} and \rightarrow_{s} potentially contradict compatibility. In the former case, the side conditions ensure that e_2 and D_2 are compatible, in the latter case compatibility follows trivially from compatibility of D_1 with e_1 .
3. *No dangling references:* Observe that only rule \rightarrow_{m} introduces a fresh location. The merge operations guarantees that this location occurs in the heap H_2 . □

From now on, we assume that configurations are well-formed, tacitly employing Lemma 4.3. 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

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

Likewise, we set $[D]_H := \{([e]_H, [\ell]_H) \mid (e, \ell) \in D\}$. Observe that $[e]_H$ is well-defined as long as H contains all locations occurring in e , similar for $[D]_H$.

Our simulation result relies on a couple of auxiliary lemmata concerning heaps. The first lemma states that in a maximally shared heap, values are stored only once.

Lemma 4.4. *Let H be a maximally shared heap with locations ℓ_1, ℓ_2 . If $[\ell_1]_H = [\ell_2]_H$ then $\ell_1 = \ell_2$.*

Proof. For a proof by contradiction, suppose $[\ell_1]_H = [\ell_2]_H$ but $\ell_1 \neq \ell_2$. Hence without loss of generality, there exist two \rightarrow_H^* -minimal nodes ℓ'_1, ℓ'_2 in H with $\ell'_1 \neq \ell'_2$ and $[\ell'_1]_H = [\ell'_2]_H$. By minimality, the latter implies that $H(\ell'_1) = H(\ell'_2)$ and this contradicts that H is maximally shared. □

The next lemma is based on the observation that the heap is monotonically increasing.

Lemma 4.5. *If $(D_1, H_1, e_1) \rightarrow_{\text{Rrsm}} (D_2, H_2, e_2)$ then the following properties hold:*

1. $[\ell]_{H_2} = [\ell]_{H_1}$ for every location ℓ of H_1 ;
2. $[D_1]_{H_2} = [D_1]_{H_1}$ and $[e_1]_{H_2} = [e_1]_{H_1}$.

Proof. As for any other step the heap remains untouched, the only non-trivial case is

$$(D_1, H_1, E[\mathbf{c}(\ell_1, \dots, \ell_k)]) \rightarrow_{\text{m}} (D_1, H_2, E[\ell]),$$

with $(H_2, \ell) = \text{merge}(H_1, \mathbf{c}(\ell_1, \dots, \ell_k))$. Observe that by definition of merge , $H_2(\ell) = H_1(\ell)$ for every $\ell \in N_{H_1}$. From this, Assertion 1 is easy to establish. Assertion 2 follows then by standard inductions on D_1 and E , respectively. □

The final lemma concerning heaps, stating that values are correctly internalized into the heap, follows essentially by definition of \rightarrow_m .

Lemma 4.6. *Let $(D, H, E[v])$ be a configuration for a value v . Then $(D, H, E[v]) \rightarrow_m^* (D, H', E[\ell])$ with $[\ell]_{H'} = [v]_H$.*

Proof. We proof the lemma by induction on the number of constructor symbols in v . In the base case $v = \ell$ the lemma trivially holds. For the inductive step, observe that $v = E'[\mathbf{c}(\ell_1, \dots, \ell_k)]$ for some evaluation context E' , and hence $(D, H, E[v]) \rightarrow_m (D, H', E[E'[\ell]])$, where $(H', \ell) = \text{merge}(H, \mathbf{c}(\ell_1, \dots, \ell_k))$. Using that $[\ell]_{H'} = [\mathbf{c}(\ell_1, \dots, \ell_k)]_{H'}$ by definition of merge and Lemma 4.5(2) we conclude $[E[E'[\ell]]]_{H'} = [E[v]]_H$. We complete this derivation to the desired form, by IH. \square

The last auxiliary lemma shows that our small-step semantics are closed under evaluation contexts.

Lemma 4.7. *Suppose $(D_1, H_1, e_1) \rightarrow (D_2, H_2, e_2)$ holds for configurations (D_1, H_1, e_1) and (D_2, H_2, e_2) , where $\rightarrow \in \{\rightarrow_R, \rightarrow_x, \rightarrow_s, \rightarrow_m\}$. Then $(D_1, H_1, E[e_1]) \rightarrow (D_2, H_2, E[e_2])$ holds for every evaluation context E such that $(D_1, H_1, E[e_1])$ is a well-formed configuration.*

Proof. The lemma can be shown by a standard induction on the context E . \square

An *initial configuration* is a well-formed configuration of the form (\emptyset, 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 . We arrive at the first crucial step in the correctness proof of the small-step semantics, with respect to the memoized operational semantics.

Lemma 4.8 (Simulation). *Let (\emptyset, H, e) be an initial configuration. If $(\emptyset, [e]_H) \Downarrow_m (C, v)$ holds for $m \geq 1$ then there exists a cache D , heap G and location ℓ in G such that $(\emptyset, H, e) \rightarrow_{R/rsm}^m (D, G, \ell)$ with $[\ell]_G = v$.*

Proof. Call a configuration (D, H, e) *proper* if it is well-formed and e does not contain a sub-expression $\langle f(v_1, \dots, v_k), e' \rangle$. We show the following claim:

Claim. For every proper configuration (D, H, e) , $([D]_H, [e]_H) \Downarrow_m (C, v)$ implies $(D, H, e) \rightarrow_{rsm}^* \cdot \rightarrow_{R/rsm}^m (D', H', \ell)$ with $([D']_{H'}, [\ell]_{H'}) = (C, v)$.

Observe that $\rightarrow_{rsm}^* \cdot \rightarrow_{R/rsm}^m = \rightarrow_{R/rsm}^m$ whenever $m > 0$. Since an initial configuration is trivially proper, the lemma follows from the claim. To prove the claim, abbreviate the relation $\rightarrow_{rsm}^* \cdot \rightarrow_{R/rsm}^m$ by \rightarrow^m for all $m \in \mathbb{N}$. Below, we tacitly employ $\rightarrow^{m_1} \cdot \rightarrow^{m_2} = \rightarrow^{m_1+m_2}$ for all $m_1, m_2 \in \mathbb{N}$. The proof is by induction on the deduction Π of the statement $([D]_H, [e]_H) \Downarrow_m (C, v)$.

- Suppose that the last rule in Π has the form:

$$\frac{\mathbf{c} \in \mathcal{C} \quad (C_{i-1}, t_i) \Downarrow_{m_i} (C_i, v_i) \quad m = \sum_{i=1}^k m_i}{(C_0, \mathbf{c}(t_1, \dots, t_k)) \Downarrow_m (C_k, \mathbf{c}(v_1, \dots, v_k))} \text{ (C-context) } .$$

Fix a proper configuration (D_0, H_0, e_0) unfolding to $(C_0, \mathbf{c}(t_1, \dots, t_k))$. Under these assumptions, either e_0 is a location or $e_0 = \mathbf{c}(e_1, \dots, e_k)$. The former case is trivial, as then t is a value and thus $m = 0$. Hence suppose $e_0 = \mathbf{c}(e_1, \dots, e_k)$. We first show that for all $i \leq k$,

$$(D_0, H_0, \mathbf{c}(e_1, \dots, e_k)) \rightarrow_{\sum_{j=1}^i m_j} (D_i, H_i, \mathbf{c}(\ell_1, \dots, \ell_i, e_{i+1}, \dots, e_k)) , \quad (\ddagger)$$

for a configuration $(D_i, H_i, \mathbf{c}(\ell_1, \dots, \ell_i, e_{i+1}, \dots, e_k))$ unfolding to $(C_i, \mathbf{c}(v_1, \dots, v_i, t_{i+1}, \dots, t_k))$. The proof is by induction on i , we consider the step from i to $i+1$. The IH yields a well-formed configuration $(D_i, H_i, E[e_{i+1}])$ for $E = \mathbf{c}(\ell_1, \dots, \ell_i, \square, e_{i+2}, \dots, e_k)$ reachable by a Derivation (\ddagger) . As the configuration (D_i, H_i, e_{i+1}) unfolds to (C_i, t_{i+1}) , the IH of the claim on the assumption $(C_i, t_{i+1}) \Downarrow_{m_{i+1}} (C_{i+1}, v_{i+1})$ yields $(D_i, H_i, e_{i+1}) \rightarrow^{m_{i+1}} (D_{i+1}, H_{i+1}, \ell_{i+1})$ where the resulting configuration unfolds to (C_{i+1}, v_{i+1}) . Iterated application of Lemma 4.7 lifts this reduction to the evaluation context E , i.e.,

$$(D_i, H_i, E[e_{i+1}]) \rightarrow^{m_{i+1}} (D_{i+1}, H_{i+1}, E[\ell_{i+1}]) .$$

Note that, with the help of Lemma 4.5(2), we can also lift the equality $[\ell_{i+1}]_{H_{i+1}} = v_{i+1}$ to

$$[E[\ell_{i+1}]]_{H_i} = \mathbf{c}(v_1, \dots, v_i, v_{i+1}, t_{i+2}, \dots, t_k).$$

As we already observed $[D_{i+1}]_{H_{i+1}} = C_{i+1}$, we conclude (‡).

In total, we thus obtain a reduction $(D_0, H_0, \mathbf{c}(e_1, \dots, e_k)) \rightarrow^m (D_k, H_k, \mathbf{c}(\ell_1, \dots, \ell_k))$ where $m = \sum_{i=1}^k m_i$ and $(D_k, H_k, \mathbf{c}(\ell_1, \dots, \ell_k))$ is a well-formed, in fact proper, configuration which unfolds to $(C_k, \mathbf{c}(v_1, \dots, v_k))$. Employing $\rightarrow^m \cdot \rightarrow_{\mathbf{m}}^* = \rightarrow^m$ we conclude the case with Lemma 4.6.

- Suppose that the last rule in Π has the form:

$$\frac{(C_{i-1}, t_i) \Downarrow_{m_i} (C_i, v_i) \quad (C_k, \mathbf{f}(v_1, \dots, v_k)) \Downarrow_n (C_{k+1}, v) \quad m = n + \sum_{i=1}^k m_i}{(C_0, \mathbf{f}(t_1, \dots, t_k)) \Downarrow_m (C_{k+1}, v)} \text{ (F-context) } .$$

Fix a proper configuration (D_0, H_0, e_0) unfolding to $(C_0, \mathbf{f}(t_1, \dots, t_k))$. By induction on k , exactly as in the previous case, we obtain a proper configuration $(D_k, H_k, \mathbf{f}(\ell_1, \dots, \ell_k))$ which unfolds to the pair $(C_k, \mathbf{f}(v_1, \dots, v_k))$ with

$$(D_0, H_0, e_0) \rightarrow^{\sum_{i=1}^k m_i} (D_k, H_k, \mathbf{f}(\ell_1, \dots, \ell_k)) .$$

The IH also yields configuration (D_{k+1}, H_{k+1}, ℓ) unfolding to (C_{k+1}, v) with

$$(D_k, H_k, \mathbf{f}(\ell_1, \dots, \ell_k)) \rightarrow^n (D_{k+1}, H_{k+1}, \ell) .$$

Summing up we conclude the case.

- Suppose that the last rule in Π has the form:

$$\frac{(\mathbf{f}(v_1, \dots, v_k), u) \in C}{(C, \mathbf{f}(v_1, \dots, v_k)) \Downarrow_0 (C, v)} \text{ (read) } .$$

Consider a proper configuration (D, H, e) that unfolds to $(C, \mathbf{f}(v_1, \dots, v_k))$. Then $e = \mathbf{f}(e_1, \dots, e_k)$, and using k applications of Lemma 4.6 we construct a reduction

$$(D, H, \mathbf{f}(e_1, \dots, e_k)) \rightarrow_{\mathbf{m}}^* (D, H_1, \mathbf{f}(\ell_1, e_2, \dots, e_k)) \rightarrow_{\mathbf{m}}^* \cdots \rightarrow_{\mathbf{m}}^* (D, H_k, \mathbf{f}(\ell_1, \dots, \ell_k)) ,$$

with $(D, H_k, \mathbf{f}(\ell_1, \dots, \ell_k))$ unfolding to $(C, \mathbf{f}(v_1, \dots, v_k))$. Lemma 4.4 and the assumption on $C = [D]_{H_k}$ implies that there exists a *unique* pair $(\mathbf{f}(\ell_1, \dots, \ell_k), \ell) \in D$ with $[\mathbf{f}(\ell_1, \dots, \ell_k)]_{H_k} = \mathbf{f}(v_1, \dots, v_k)$ and $[\ell]_{H_k} = v$. Thus overall

$$(D, H, e) = (D, H, \mathbf{f}(e_1, \dots, e_k)) \rightarrow_{\mathbf{m}}^* (D, H_k, \mathbf{f}(\ell_1, \dots, \ell_k)) \rightarrow_{\mathbf{r}} (D, H_k, \ell) ,$$

where (D, H_k, ℓ) unfolds to (C, v) . Using $\rightarrow_{\mathbf{m}}^* \cdot \rightarrow_{\mathbf{r}} \subseteq \rightarrow^0$ we conclude the case.

- Finally, suppose that the last rule in Π has the form:

$$\frac{(\mathbf{f}(v_1, \dots, v_k), v) \notin C \quad \mathbf{f}(p_1, \dots, p_k) \rightarrow r \in \mathcal{R} \quad \forall i. p_i \sigma = v_i \quad (C, r\sigma) \Downarrow_m (C', v)}{(C, \mathbf{f}(v_1, \dots, v_k)) \Downarrow_{m+1} (C' \cup \{(\mathbf{f}(v_1, \dots, v_k), v)\}, v)} \text{ (update) } .$$

Fix a proper configuration (D, H, e) that unfolds to $(C, \mathbf{f}(v_1, \dots, v_k))$, in particular $e = \mathbf{f}(v_1, \dots, v_k)$. As above, we see $(D, H, e) \rightarrow_{\mathbf{m}}^* (D, H_k, \mathbf{f}(\ell_1, \dots, \ell_k))$ for a configuration $(D, H_k, \mathbf{f}(\ell_1, \dots, \ell_k))$ also unfolding to $(C, \mathbf{f}(v_1, \dots, v_k))$. Suppose that the cache D maps $\mathbf{f}(\ell_1, \dots, \ell_k)$ to some location ℓ . But then $\mathbf{f}(v_1, \dots, v_k)$ would be cached in C , contradicting our assumption. Thus we conclude $(\mathbf{f}(\ell_1, \dots, \ell_k), \ell) \notin D$, for all locations ℓ . Since Proposition 3.9 on the assumption yields $\triangle(\mathbf{f}(\ell_1, \dots, \ell_k)) \geq_m \mathbf{f}(H \setminus \ell_1, \dots, H \setminus \ell_k)$ for a matching morphism m , in total we obtain

$$(D, H, e) \rightarrow_{\mathbf{m}}^* (D, H, \mathbf{f}(\ell_1, \dots, \ell_k)) \rightarrow_{\mathbf{R}} (D, H_k, \langle \mathbf{f}(\ell_1, \dots, \ell_k), r\sigma_m \rangle) .$$

Note that by Proposition 3.9 the substitution σ and induced substitution σ_m satisfy $\sigma(x) = [\sigma_m(x)]_{H_k}$ for all variables x in r . Hence by a standard induction on r , $[r\sigma_m]_{H_k} = r\sigma$ follows. We conclude that $(D, H_k, \langle f(\ell_1, \dots, \ell_k), r\sigma_m \rangle)$ unfolds to $(C, r\sigma)$. The IH yields a well-formed configuration (D', G, ℓ) unfolding to (C', v) with $(D, H_k, r\sigma_m) \rightarrow^m (D', G, \ell)$. Thus

$$\begin{aligned} (D, H_k, \langle f(\ell_1, \dots, \ell_k), r\sigma_m \rangle) &\rightarrow^m (D', G, \langle f(\ell_1, \dots, \ell_k), \ell \rangle) \\ &\rightarrow_{\mathfrak{s}} (D' \cup \{(\langle f(\ell_1, \dots, \ell_k), \ell \rangle), G, \ell\}). \end{aligned}$$

Using that $[\ell]_G = v$ and $[f(\ell_1, \dots, \ell_k)]_{H_k} = f(v_1, \dots, v_k)$, Lemma 4.5 yields

$$[D' \cup \{(\langle f(\ell_1, \dots, \ell_k), \ell \rangle), G, \ell\}]_G = C' \cup \{(\langle f(v_1, \dots, v_k), v \rangle)\}.$$

Putting things together, employing $\rightarrow_{\mathfrak{m}}^* \cdot \rightarrow_{\mathfrak{R}} \subseteq \rightarrow^1$ and $\rightarrow^m \cdot \rightarrow_{\mathfrak{s}} = \rightarrow^m$ we obtain

$$(D, H, e) \rightarrow^{m+1} (D' \cup \{(\langle f(\ell_1, \dots, \ell_k), \ell \rangle), G, \ell\}),$$

and conclude this final case. \square

The following lemma completes the proof of correctness of $\rightarrow_{\mathfrak{Rrsm}}$. Here, a binary 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 4.9 (Determinism).

1. The relations $\rightarrow_{\mathfrak{R}}$, $\rightarrow_{\mathfrak{r}}$, $\rightarrow_{\mathfrak{s}}$ and $\rightarrow_{\mathfrak{m}}$ are deterministic on well-formed configurations.
2. The relation $\rightarrow_{\mathfrak{Rrsm}}$ is deterministic on well-formed configurations.

Proof. We first consider Assertion 1. Fix a well-formed configuration (D, H, e) . We show that for $\rightarrow \in \{\rightarrow_{\mathfrak{R}}, \rightarrow_{\mathfrak{r}}, \rightarrow_{\mathfrak{s}}, \rightarrow_{\mathfrak{m}}\}$, any peak $(D_1, H_1, e_1) \leftarrow (D, H, e) \rightarrow (D_2, H_2, e_2)$ is *trivial*, i.e. $(D_1, H_1, e_1) = (D_2, H_2, e_2)$. Notice that the expression e is uniquely decomposed into an evaluation context E and redex e' with $e = E[e']$. By definition it is thus immediate that the relations $\rightarrow_{\mathfrak{s}}$ and $\rightarrow_{\mathfrak{m}}$ are deterministic. Similar, determinism of the relation $\rightarrow_{\mathfrak{r}}$ follows from the assumption that the configuration (D, H, e) is wellformed, in particular, that the cache D is a function. Finally, consider a peak with the relation $\rightarrow_{\mathfrak{R}}$. Again this case follows from the unique decomposition of e into evaluation context and redex, together with the non-overlapping condition imposed on rules in \mathcal{R} , tacitly employing Proposition 3.9.

Finally, to prove Assertion 2 we consider a peak $(D_1, H_1, e_1) \leftarrow_a (D, H, e) \rightarrow_b (D_2, H_2, e_2)$ for two relations $\rightarrow_a, \rightarrow_b \in \{\rightarrow_{\mathfrak{R}}, \rightarrow_{\mathfrak{r}}, \rightarrow_{\mathfrak{s}}, \rightarrow_{\mathfrak{m}}\}$. We show that the considered peak is trivial. Due to the first assertion, it suffices to consider only the cases where $\rightarrow_a \neq \rightarrow_b$. The proof is by induction on the expression e . There are three base cases to consider: (i) $e = f(\ell_1, \dots, \ell_k)$, (ii) $e = \langle f(\ell_1, \dots, \ell_k), \ell \rangle$ and (iii) $e = \mathbf{c}(\ell_1, \dots, \ell_k)$. The only potential peak can occur in case (i) with relations $\rightarrow_{\mathfrak{R}}$ and $\rightarrow_{\mathfrak{r}}$. However, any peak is excluded in this case with the pre-conditions put on the heap H . We conclude the base cases. For the inductive step, we consider a peak

$$(D_1, H_1, E[e'_1]) \leftarrow_a (D, H, E[e']) \rightarrow_b (D_2, H_2, E[e'_2]),$$

where $e = E[e']$ for a context E . We thus have a peak $(D_1, H_1, e'_1) \leftarrow_a (D, H, e') \rightarrow_b (D_2, H_2, e'_2)$, which by IH is trivial. We conclude the inductive step. \square

Theorem 4.10. *Suppose $(\emptyset, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$ holds for a reducible term $f(v_1, \dots, v_k)$. Then for each initial configuration (\emptyset, H, e) with $[e]_H = f(v_1, \dots, v_k)$, there exists a unique sequence $(\emptyset, H, e) \rightarrow_{\mathfrak{R}/\mathfrak{rsm}}^m (D, G, \ell)$ 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 4.8 and Lemma 4.9. \square

4.2. Invariance

Theorem 4.10 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 steps 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 e 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, the *weight* $\text{wt}(e)$ is defined by ignoring locations, i.e. $\text{wt}(\ell) := 0$. Recall that a reduction $(D_1, H_1, e_1) \rightarrow_{\mathbb{R}/\text{rsm}}^m (D_2, H_2, e_2)$ consists of m applications of $\rightarrow_{\mathbb{R}}$, all possibly interleaved by \rightarrow_{rsm} -reductions. As a first step, we thus estimate the overall length of the reduction $(D_1, H_1, e_1) \rightarrow_{\mathbb{R}/\text{rsm}}^m (D_2, H_2, e_2)$ 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 4.11. *The following properties hold:*

1. If $(D_1, H_1, e_1) \rightarrow_{\text{rsm}} (D_2, H_2, e_2)$ then $\text{wt}(e_2) < \text{wt}(e_1)$.
2. If $(D_1, H_1, e_1) \rightarrow_{\mathbb{R}} (D_2, H_2, e_2)$ then $\text{wt}(e_2) \leq \text{wt}(e_1) + \Delta$.

Proof. The first assertion follows by case analysis on \rightarrow_{rsm} . For the second, suppose $(D_1, H_1, e_1) \rightarrow_{\mathbb{R}} (D_2, H_2, e_2)$ where $e_1 = E[f(\ell_1, \dots, \ell_k)]$ and $e_2 = E[\langle f(\ell_1, \dots, \ell_k), r\sigma_m \rangle]$ for a rule $f(l_1, \dots, l_k) \rightarrow r \in \mathcal{R}$. Observe that since the substitution σ_m replaces variables by locations, $\Delta \geq |r| = |r\sigma_m| \geq \text{wt}(r\sigma_m)$ holds. Consequently,

$$\text{wt}(f(\ell_1, \dots, \ell_k)) + \Delta \geq 1 + \text{wt}(r\sigma_m) = \text{wt}(\langle f(\ell_1, \dots, \ell_k), r\sigma_m \rangle).$$

From this, the assertion follows by a standard induction on E . □

Then essentially an application of the *weight gap principle* [20], a form of *amortized cost analysis*, binds the overall length of an $\rightarrow_{\mathbb{R}/\text{rsm}}^m$ -reduction suitably.

Lemma 4.12. *If $(D_1, H_1, e_1) \rightarrow_{\mathbb{R}/\text{rsm}}^m (D_2, H_2, e_2)$ then $(D_1, H_1, e_1) \rightarrow_{\text{Rrsm}}^n (D_2, H_2, e_2)$ for $n \leq (1 + \Delta) \cdot m + \text{wt}(e)$ and $\Delta \in \mathbb{N}$ a constant depending only on \mathcal{P} .*

Proof. For a configuration $c = (D, H, e)$ define $\text{wt}(c) := \text{wt}(e)$ and let Δ be defined as in Lemma 4.11. Consider $(D_1, H_1, e_1) \rightarrow_{\mathbb{R}/\text{rsm}}^m (D_2, H_2, e_2)$, which can be written as a reduction

$$(D_1, H_1, e_1) = c_0 \xrightarrow{\text{rsm}}^{n_0} d_0 \rightarrow_{\mathbb{R}} c_1 \xrightarrow{\text{rsm}}^{n_1} d_1 \rightarrow_{\mathbb{R}} \dots \xrightarrow{\text{rsm}}^{n_m} d_m, \quad (\ddagger)$$

of length $n := m + \sum_{i=0}^k n_i$. Lemma 4.11 yields (i) $n_i \leq \text{wt}(c_i) - \text{wt}(d_i)$ for all $0 \leq i \leq m$; and (ii) $\text{wt}(c_{i+1}) - \text{wt}(d_i) \leq \Delta$ for all $0 \leq i < m$. Hence overall, the Reduction (\ddagger) is of length

$$\begin{aligned} n &\leq m + (\text{wt}(c_0) - \text{wt}(d_0)) + \dots + (\text{wt}(c_m) - \text{wt}(d_m)) \\ &= m + \text{wt}(c_0) + (\text{wt}(c_1) - \text{wt}(d_0)) + \dots + (\text{wt}(c_m) - \text{wt}(d_{m-1})) - \text{wt}(d_m) \\ &\leq m + \text{wt}(c_0) + m \cdot \Delta \\ &= (1 + \Delta) \cdot m + \text{wt}(e). \end{aligned}$$

The lemma follows. □

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, similarly, 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 4.13. *If $(D_1, H_1, e_1) \rightarrow_{\text{Rrsm}} (D_2, H_2, e_2)$ then $|(D_2, H_2, e_2)| \leq |(D_1, H_1, e_1)| + \Delta$.*

Proof. The lemma follows by case analysis on the rule applied in $(D_1, H_1, e_1) \rightarrow_{\text{Rrsm}} (D_2, H_2, e_2)$, using $1 \leq \Delta$. \square

Theorem 4.14. *There exists a polynomial $p : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ such that for every initial configuration (\emptyset, H_1, e_1) , a configuration (D_2, H_2, e_2) with $(\emptyset, H_1, e_1) \xrightarrow{m}_{\text{R/rsm}} (D_2, H_2, e_2)$ is computable from (\emptyset, 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 $c \rightarrow_{\text{Rrsm}} d$, i.e. which maps $[c]$ to $[d]$, is computable in polynomial time in $[c]$, and thus in the size $|c|$ of the configuration c . Iterating this function at most $n := (1 + \Delta) \cdot m + |(\emptyset, H_1, e_1)|$ times on input $[(\emptyset, H_1, e_1)]$, yields the desired result $[(D_2, H_2, e_2)]$ by Lemma 4.12. Since each iteration increases the size of a configuration by at most the constant Δ (Lemma 4.13), in particular the size of each intermediate configuration is bounded by a linear function in $|(\emptyset, H_1, e_1)| = |H_1| + |e_1|$ and n , the theorem follows. \square

Combining Theorem 4.10 and Theorem 4.14 we thus obtain the desired invariance result.

Theorem 4.15 (Invariance of Memoized Runtime Complexity). *There exists a polynomial $p : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ such that for $(\emptyset, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$, the value v represented as a graph is computable from v_1, \dots, v_k in time $p(\sum_{i=1}^k |v_i|, m)$.*

Theorem 4.15 thus confirms that the cost m of a reduction $(\emptyset, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$ 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. $(\emptyset, f(v_1, \dots, v_k)) \Downarrow_m (C, v)$ with $\sum_{i=1}^k |v_i| \leq n$, is an *invariant cost model*.

Example 4.16 (Continued from Example 3.6). Reconsider the program $\text{P}_{\text{rabbits}}$ and the evaluation of a call $\mathbf{r}(\mathbf{S}^n(\mathbf{0}))$ 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 $\mathbf{r}(\mathbf{S}^n(\mathbf{0})) \Downarrow_m v_n$ with $m \leq 2 \cdot n + 1$. Crucially here, the two intermediate functions \mathbf{a} and \mathbf{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 \mathbf{a} and \mathbf{b} , respectively, are unfolded at most n times. As a consequence of the bound on m and Theorem 4.15 we obtain that the function `rabbits` from the introduction is polytime computable.

Remark 3. Strictly speaking, our graph 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 more compact than a linear representation without sharing, it is not exponentially more *succinct* than a reasonable encoding for graphs (e.g. representations as circuits). 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, in the size of the given DAGs. A very simple algorithm that compares two term graphs for equality and that runs in time linear in the number of nodes of the two DAGs is given e.g. in [21, Proposition 4]. Exploiting that the graphs under consideration are rooted, the algorithm realizes the comparison through a node traversal inverse to the topological order, employing dynamic programming techniques to get to the sharp complexity bound.

5. GRSR is Sound for Polynomial Time

Sometimes (e.g., in [13]), 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 a 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 $\text{STs}(v_1, \dots, v_k) := \bigcup_{1 \leq i \leq k} \text{STs}(v_i)$. Note that $\|v_1, \dots, v_k\|$ corresponds to the number of locations necessary to store the values v_1, \dots, v_k on a heap (compare Lemma 4.4). If \mathbf{A} is the expression $\mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_k}$, n is a natural number, and \vec{t} is a sequence of k terms, then $\vec{t} \upharpoonright_{\mathbf{A}}^{>n}$ is defined to be t_{i_1}, \dots, t_{i_l} , where i_1, \dots, i_l are precisely those indices such that $p_{i_1}, \dots, p_{i_l} > n$. Similarly $\vec{t} \upharpoonright_{\mathbf{A}}^{=n}$, and expressions like $\|\vec{t}\|_{\mathbf{A}}^{>n}$ and $\|\vec{t}\|_{\mathbf{A}}^{=n}$, are defined. For a function $\mathbf{f} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$ and term $\mathbf{f}(\vec{t})$, the arguments $\vec{t} \upharpoonright_{\mathbf{A}}^{>n}$ thus precisely correspond to the arguments in \vec{t} of tier higher than the output tier n of \mathbf{f} , i.e. those that are substituted in potential recurrence positions of \mathbf{f} . Likewise, $\vec{t} \upharpoonright_{\mathbf{A}}^{=n}$ singles out those arguments in \vec{t} that potentially contribute to the output of $\mathbf{f}(\vec{t})$, but which are not in recurrence position. The expressions $\|\vec{t}\|_{\mathbf{A}}^{>n}$ and $\|\vec{t}\|_{\mathbf{A}}^{=n}$ refer then to the minimal shared size of the arguments $\vec{t} \upharpoonright_{\mathbf{A}}^{>n}$ and $\vec{t} \upharpoonright_{\mathbf{A}}^{=n}$, respectively.

Lemma 5.1 (Max-Poly). *If $\mathbf{f} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$, then there is a polynomial $p_{\mathbf{f}} : \mathbb{N} \rightarrow \mathbb{N}$ such that $\|\mathbf{f}(\vec{v})\| \leq \|\vec{v}\|_{\mathbf{A}}^{=n} + p_{\mathbf{f}}(\|\vec{v}\|_{\mathbf{A}}^{>n})$ holds for all \vec{v} .*

Proof. Recall that for a sequence of functions $\vec{\mathbf{f}} = \mathbf{f}_1, \dots, \mathbf{f}_k$ we denote by $\vec{\mathbf{f}} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$ that $\mathbf{f}_i \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$ is derivable for all $i \in \{1, \dots, k\}$. Let us abbreviate with $\vec{\mathbf{f}}(\vec{t})$ the sequence of terms $\mathbf{f}_1(\vec{t}), \dots, \mathbf{f}_k(\vec{t})$. We prove the following strengthening of the statement: for every sequence of functions $\vec{\mathbf{f}}$ with $\vec{\mathbf{f}} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$, there is a monotone polynomial $p_{\vec{\mathbf{f}}}$ such that for all terms \vec{t} there exists a set $C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}}$ such that

$$\text{STs}(\vec{\mathbf{f}}(\vec{t})) \subseteq \text{STs}(\vec{t} \upharpoonright_{\mathbf{A}}^{=n}) \cup C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}} \quad \text{and} \quad |C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}}| \leq p_{\vec{\mathbf{f}}}(\|\vec{t}\|_{\mathbf{A}}^{>n}). \quad (\ddagger)$$

The set $\text{STs}(\vec{t} \upharpoonright_{\mathbf{A}}^{=n})$ accounts for those subterms of the results that are copies of the input, whereas $C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}}$ accounts for those subterms that are *created*. The proof is by induction on the maximum of depths of the individual derivations in $\vec{\mathbf{f}} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$.

Observe that if the claim (\ddagger) holds for two sequences of functions $\vec{\mathbf{f}}$ and $\vec{\mathbf{g}}$, then the claim (\ddagger) also holds for the concatenated sequence $\vec{\mathbf{f}}, \vec{\mathbf{g}}$. To see this, fix arguments \vec{t} , define $C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}, \vec{\mathbf{g}}} := C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}} \cup C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{g}}}$ and $p_{\vec{\mathbf{f}}, \vec{\mathbf{g}}}(x) := p_{\vec{\mathbf{f}}}(x) + p_{\vec{\mathbf{g}}}(x)$. Then

$$\text{STs}(\vec{\mathbf{f}}(\vec{t}), \vec{\mathbf{g}}(\vec{t})) = \text{STs}(\vec{\mathbf{f}}(\vec{t})) \cup \text{STs}(\vec{\mathbf{g}}(\vec{t})) \subseteq \text{STs}(\vec{t} \upharpoonright_{\mathbf{A}}^{=n}) \cup C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}} \cup \text{STs}(\vec{t} \upharpoonright_{\mathbf{A}}^{=n}) \cup C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{g}}} = \text{STs}(\vec{t} \upharpoonright_{\mathbf{A}}^{=n}) \cup C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}, \vec{\mathbf{g}}},$$

and likewise

$$|C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}, \vec{\mathbf{g}}}| = |C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{f}}} \cup C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\vec{\mathbf{g}}}| \leq p_{\vec{\mathbf{f}}}(\|\vec{t}\|_{\mathbf{A}}^{>n}) + p_{\vec{\mathbf{g}}}(\|\vec{t}\|_{\mathbf{A}}^{>n}) = p_{\vec{\mathbf{f}}, \vec{\mathbf{g}}}(\|\vec{t}\|_{\mathbf{A}}^{>n}).$$

Indeed, it is thus justified to consider elements \mathbf{f} of $\vec{\mathbf{f}}$ individually, and combine the resulting sets $C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\mathbf{f}}$ and polynomials $p_{\mathbf{f}}$ according to the above recipe. We continue with the proof of (\ddagger) by case analysis on the functions \mathbf{f} occurring in the sequence $\vec{\mathbf{f}}$.

- If \mathbf{f} is a constructor function $\mathbf{f}_{\mathbf{c}}$, where $\mathbf{f}_{\mathbf{c}} \triangleright \mathbb{A}_n^{\text{ar}(\mathbf{c})} \rightarrow \mathbb{A}_n$, $\vec{t} \upharpoonright_{\mathbf{A}}^{>n}$ is the empty sequence ϵ and $\vec{t} \upharpoonright_{\mathbf{A}}^{=n} = \vec{t}$, we define $C_{\epsilon}^{\mathbf{f}_{\mathbf{c}}} := \{\mathbf{c}(\vec{t})\}$ and $p_{\mathbf{f}_{\mathbf{c}}} := 1$. Using that $\text{STs}(\mathbf{f}_{\mathbf{c}}(\vec{t})) = \{\mathbf{c}(\vec{t})\} \cup \text{STs}(\vec{t})$ we conclude the claim (\ddagger) for this case. In a similar spirit, we handle the case where \mathbf{f} is a projection function Π_l^k .
- If \mathbf{f} is defined by case distinction, we conclude directly from the induction hypothesis.
- Suppose \mathbf{f} is defined by composition. For the sake of readability, let us consider the special case $\mathbf{f} = \mathbf{g} \circ (\mathbf{h}_1, \mathbf{h}_2)$ with $\mathbf{f} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$ and $\mathbf{g} \triangleright \mathbb{A}_m \times \mathbb{A}_m \rightarrow \mathbb{A}_n$ where $m > n$, and thus $\mathbf{h}_1 \triangleright \mathbf{A} \rightarrow \mathbb{A}_m$ and $\mathbf{h}_2 \triangleright \mathbf{A} \rightarrow \mathbb{A}_m$. Note that the IH is applicable to \mathbf{g} , \mathbf{h}_1 and \mathbf{h}_2 , which yields appropriate polynomials $p_{\mathbf{g}}$, $p_{\mathbf{h}_1}$ and $p_{\mathbf{h}_2}$ together with sets $C_t^{\mathbf{g}}$, $C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\mathbf{h}_1}$ and $C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\mathbf{h}_2}$ for all terms $t, \vec{t} \upharpoonright_{\mathbf{A}}^{>n}$. Let us then define

$$C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\mathbf{f}} := C_{\vec{t} \upharpoonright_{\mathbf{A}}^{>n}}^{\mathbf{h}_2} \cup C_{\mathbf{h}_1(\vec{t})}^{\mathbf{g}} \quad \text{and} \quad p_{\mathbf{f}}(x) := p_{\mathbf{h}_2}(x) + p_{\mathbf{g}}(x + p_{\mathbf{h}_1}(x)).$$

Then we have:

$$\begin{aligned}
|C_{\vec{t} \rhd_{\mathbf{A}}}^{\mathbf{f}}| &\leq |C_{\vec{t} \rhd_{\mathbf{A}}}^{\mathbf{h}_2}| + |C_{\mathbf{h}_1(\vec{t})}^{\mathbf{g}}| \leq p_{\mathbf{h}_2}(\|\vec{t}\|_{\mathbf{A}}^{\geq n}) + p_{\mathbf{g}}(\|\mathbf{h}_1(\vec{t})\|) \\
&\leq p_{\mathbf{h}_2}(\|\vec{t}\|_{\mathbf{A}}^{\geq n}) + p_{\mathbf{g}}(\|\vec{t}\|_{\mathbf{A}}^{\leq m} + p_{\mathbf{h}_1}(\|\vec{t}\|_{\mathbf{A}}^{\geq m})) \\
&\leq p_{\mathbf{h}_2}(\|\vec{t}\|_{\mathbf{A}}^{\geq n}) + p_{\mathbf{g}}(\|\vec{t}\|_{\mathbf{A}}^{\geq n} + p_{\mathbf{h}_1}(\|\vec{t}\|_{\mathbf{A}}^{\geq n})) \\
&= p_{\mathbf{f}}(\|\vec{t}\|_{\mathbf{A}}^{\geq n}) .
\end{aligned}$$

Note that for the last inequality, we use the assumption $m > n$. As moreover:

$$\begin{aligned}
\text{STs}(\vec{\mathbf{f}}(\vec{t})) &= \text{STs}(\mathbf{g}(\mathbf{h}_1(\vec{t}), \mathbf{h}_2(\vec{t}))) \subseteq \text{STs}(\mathbf{h}_2(\vec{t})) \cup C_{\mathbf{h}_1(\vec{t})}^{\mathbf{g}} \\
&\subseteq \text{STs}(\vec{t} \downarrow_{\mathbf{A}}^{\leq n}) \cup C_{\vec{t} \rhd_{\mathbf{A}}}^{\mathbf{h}_2} \cup C_{\mathbf{h}_1(\vec{t})}^{\mathbf{g}} \\
&= \text{STs}(\vec{t} \downarrow_{\mathbf{A}}^{\leq n}) \cup C_{\vec{t} \rhd_{\mathbf{A}}}^{\mathbf{f}} ,
\end{aligned}$$

the case follows.

- The interesting case is the one in which the function \mathbf{f} in $\vec{\mathbf{f}}$ is obtained through a single instance of simultaneous primitive recursion. For the sake of readability, let us consider the case where $\mathbf{f} = \mathbf{f}_1$ and \mathbf{f}_2 are defined by general ramified simultaneous recurrence over binary trees as in (†) on page 4, and where $\mathbf{A} = \mathbb{T}_1 \times \mathbb{T}_1 \times \mathbb{T}_0$ and $n = 0$, i.e.

$$\mathbf{f}_1, \mathbf{f}_2 \triangleright \mathbb{T}_1 \times \mathbb{T}_1 \times \mathbb{T}_0 \rightarrow \mathbb{T}_0 \quad \mathbf{g}_1, \mathbf{g}_2 \triangleright \mathbb{T}_1 \times \mathbb{T}_0 \rightarrow \mathbb{T}_0 \quad \mathbf{h}_1, \mathbf{h}_2 \triangleright \mathbb{T}_1^3 \times \mathbb{T}_0^5 \rightarrow \mathbb{T}_0 .$$

Generalization of this case is straight forward. Let us abbreviate $\vec{\mathbf{g}} = \mathbf{g}_1, \mathbf{g}_2$ and $\vec{\mathbf{h}} = \mathbf{h}_1, \mathbf{h}_2$. Note that the height of the tiering derivations of $\vec{\mathbf{g}}$ and $\vec{\mathbf{h}}$ is strictly smaller than the one of the considered function \mathbf{f}_1 . Thus the IH is applicable to both $\vec{\mathbf{g}}$ and $\vec{\mathbf{h}}$, from which we obtain appropriate polynomials $p_{\vec{\mathbf{g}}}$ and $p_{\vec{\mathbf{h}}}$ as well as sets $C_t^{\vec{\mathbf{g}}}$ and $C_{r,s,t}^{\vec{\mathbf{h}}}$ for all terms r, s, t . Let us then define

$$C_{s,t}^{\mathbf{f}_1, \mathbf{f}_2} := C_t^{\vec{\mathbf{g}}} \cup \bigcup_{\mathbf{bin}(r,q) \in \text{STs}(s)} C_{r,q,t}^{\vec{\mathbf{h}}} \quad \text{and} \quad p_{\mathbf{f}_1, \mathbf{f}_2}(x) := p_{\vec{\mathbf{g}}}(x) + x \cdot p_{\vec{\mathbf{h}}}(x) .$$

First of all, one easily realizes that:

$$\begin{aligned}
|C_{s,t}^{\mathbf{f}_1, \mathbf{f}_2}| &\leq |C_t^{\vec{\mathbf{g}}}| + \left| \bigcup_{\mathbf{bin}(r,q) \in \text{STs}(s)} C_{r,q,t}^{\vec{\mathbf{h}}} \right| \leq p_{\vec{\mathbf{g}}}(\|t\|) + \sum_{\mathbf{bin}(r,q) \in \text{STs}(s)} p_{\vec{\mathbf{h}}}(\|r, q, t\|) \leq p_{\vec{\mathbf{g}}}(\|t\|) + \sum_{\mathbf{bin}(r,q) \in \text{STs}(s)} p_{\vec{\mathbf{h}}}(\|\mathbf{bin}(r, q), t\|) \\
&\leq p_{\vec{\mathbf{g}}}(\|t\|) + \sum_{\mathbf{bin}(r,q) \in \text{STs}(s)} p_{\vec{\mathbf{h}}}(\|s, t\|) \leq p_{\vec{\mathbf{g}}}(\|t\|) + \|s\| \cdot p_{\vec{\mathbf{h}}}(\|s, t\|) \\
&\leq p_{\mathbf{f}_1, \mathbf{f}_2}(\|s, t\|) .
\end{aligned}$$

Moreover, by induction on terms, also the other requirement is satisfied:

$$\begin{aligned}
\text{STs}(\mathbf{f}_i(\mathbf{nil}, s, t)) &= \text{STs}(\mathbf{g}_i(s, t)) \subseteq \text{STs}(t) \cup C_s^{\vec{\mathbf{g}}} = \text{STs}(t) \cup C_{\mathbf{nil}, s}^{\mathbf{f}_1, \mathbf{f}_2} ; \\
\text{STs}(\mathbf{f}_i(\mathbf{bin}(r, q), s, t)) &= \text{STs}(\mathbf{h}_i(r, q, s, t, \mathbf{f}_1(r, s, t), \mathbf{f}_1(q, s, t), \mathbf{f}_2(r, s, t), \mathbf{f}_2(q, s, t))) \\
&\subseteq \text{STs}(t) \cup \text{STs}(\mathbf{f}_1(r, s, t), \mathbf{f}_2(r, s, t)) \cup \text{STs}(\mathbf{f}_1(q, s, t), \mathbf{f}_2(q, s, t)) \cup C_{r,q,s}^{\vec{\mathbf{h}}} \\
&\subseteq \text{STs}(t) \cup \text{STs}(t) \cup C_{r,s}^{\mathbf{f}_1, \mathbf{f}_2} \cup \text{STs}(t) \cup C_{q,s}^{\mathbf{f}_1, \mathbf{f}_2} \cup C_{r,q,s}^{\vec{\mathbf{h}}} \\
&= \text{STs}(t) \cup C_{r,s}^{\mathbf{f}_1, \mathbf{f}_2} \cup C_{q,s}^{\mathbf{f}_1, \mathbf{f}_2} \cup C_{r,q,s}^{\vec{\mathbf{h}}} \\
&= \text{STs}(t) \cup C_s^{\vec{\mathbf{g}}} \cup \left(\bigcup_{\mathbf{bin}(p,o) \in \text{STs}(r)} C_{p,o,s}^{\vec{\mathbf{h}}} \right) \cup C_s^{\vec{\mathbf{g}}} \cup \left(\bigcup_{\mathbf{bin}(p,o) \in \text{STs}(q)} C_{p,o,s}^{\vec{\mathbf{h}}} \right) \cup C_{r,q,s}^{\vec{\mathbf{h}}} \\
&\subseteq \text{STs}(t) \cup C_s^{\vec{\mathbf{g}}} \cup \bigcup_{\mathbf{bin}(p,o) \in \text{STs}(\mathbf{bin}(r,q))} C_{p,o,s}^{\vec{\mathbf{h}}} \\
&= \text{STs}(t) \cup C_{\mathbf{bin}(r,q), s}^{\mathbf{f}_1, \mathbf{f}_2} .
\end{aligned}$$

This concludes the proof. \square

Once we know that ramified recursive definitions are not too fast-growing with respect to the minimal shared size, we know that all terms around do not have a too-big minimal shared size. There is still some work to be done to get to our goal. It is convenient to introduce another auxiliary concept before proceeding. If \mathbf{f} is a function on n arguments in $\text{SIMREC}(\mathbb{A})$ and v_1, \dots, v_n are elements of \mathbb{A} , we define the set $FC_{v_1, \dots, v_n}^{\mathbf{f}}$ as follows. The set $FC_{v_1, \dots, v_n}^{\mathbf{f}}$ captures *function calls* that arise in the computation of $\mathbf{f}(v_1, \dots, v_n)$. The definition is by induction on the definition of \mathbf{f} .

$$\begin{aligned}
FC_{v_1, \dots, v_n}^{\mathbf{f}_c} &= \{(\mathbf{f}_c, v_1, \dots, v_n)\}; \\
FC_{v_1, \dots, v_n}^{\Pi_n^m} &= \{(\Pi_n^m, v_1, \dots, v_n)\}; \\
FC_{v_1, \dots, v_n}^{\mathbf{f} \circ (\mathbf{g}_1, \dots, \mathbf{g}_m)} &= \bigcup_{i=1}^m FC_{v_1, \dots, v_n}^{\mathbf{g}_i} \cup FC_{\mathbf{g}_1(\vec{v}), \dots, \mathbf{g}_m(\vec{v})}^{\mathbf{f}} \cup \{(\mathbf{f} \circ (\mathbf{g}_1, \dots, \mathbf{g}_m), v_1, \dots, v_n)\}; \\
FC_{\mathbf{c}_i(u_1, \dots, u_{\text{ar}(\mathbf{c}_i)}), v_2, \dots, v_n}^{\text{case}((\mathbf{f}_i)_{1 \leq i \leq k})} &= FC_{u_1, \dots, u_{\text{ar}(\mathbf{c}_i)}, v_2, \dots, v_n}^{\mathbf{f}_i} \cup \{(\text{case}((\mathbf{f}_i)_{1 \leq i \leq k}), \mathbf{c}_i(u_1, \dots, u_{\text{ar}(\mathbf{c}_i)}), v_2, \dots, v_n)\}; \\
FC_{\mathbf{c}_i(u_1, \dots, u_{\text{ar}(\mathbf{c}_i)}), v_2, \dots, v_n}^{\mathbf{g}_j} &= \bigcup_{i=1}^{\text{ar}(\mathbf{c}_i)} \bigcup_{j=1}^m FC_{u_i, v_2, \dots, v_n}^{\mathbf{g}_j} \cup FC_{u_1, \dots, u_{\text{ar}(\mathbf{c}_i)}, v_2, \dots, v_n, \mathbf{g}_1(u_1, v_2, \dots, v_n), \dots, \mathbf{g}_m(u_{\text{ar}(\mathbf{c}_i)}, v_2, \dots, v_n)}^{\mathbf{f}_{i,j}} \\
&\quad \cup \{(\mathbf{g}_j, \mathbf{c}_i(u_1, \dots, u_{\text{ar}(\mathbf{c}_i)}), v_2, \dots, v_n)\} \\
&\quad \text{where } \mathbf{g}_j = \text{simrec}((\mathbf{f}_{i,j})_{1 \leq i \leq k, 1 \leq j \leq m}).
\end{aligned}$$

Please observe how subsets of $FC_{\vec{v}}^{\mathbf{f}}$ are in natural correspondence to (proper) caches for the program $P_{\mathbf{f}}$. Justified by this observation, we identify sets $FC_{\vec{v}}^{\mathbf{f}}$ with caches $\{(g(\vec{v}), u) \mid (g, \vec{v}) \in FC_{\vec{v}}^{\mathbf{f}} \text{ and } f(\vec{v}) \downarrow u\}$. The parameter we have just introduced is all that is needed to estimate the memoized runtime complexity.

Lemma 5.2. *If \mathbf{f} is a function on k arguments in $\text{SIMREC}(\mathbb{A})$, then for every $v_1, \dots, v_k \in \mathbb{A}$ and for every proper cache C , there exists a cache D , value u and natural number $m \leq |FC_{\vec{v}}^{\mathbf{f}}|$ such that $(C, f(v_1, \dots, v_k)) \Downarrow_m (D, u)$ with respect to the program $P_{\mathbf{f}}$ associated with \mathbf{f} .*

Proof. We prove the lemma by induction on the proof that \mathbf{f} is in $\text{SIMREC}(\mathbb{A})$. We show the stronger claim that $m \leq |FC_{\vec{v}}^{\mathbf{f}} \setminus C|$ holds and that $D := C \cup FC_{\vec{v}}^{\mathbf{f}}$. We consider the most interesting case where \mathbf{f} is defined by general ramified simultaneous recurrence. As before, for brevity we treat only the special case where two ternary functions $\vec{\mathbf{f}} = \mathbf{f}_1, \mathbf{f}_2$ over binary trees are defined simultaneously, as in (†) on page 4. Generalization of this case is straight forward. Let C be a proper cache. The case itself is proven by structural induction on the recursion parameter.

- In the base case, we consider the statement $(C, f_i(\mathbf{nil}, \vec{v})) \Downarrow_m (D, u)$. Note that the program $P_{\mathbf{f}}$ contains the rule $f_i(\mathbf{nil}, \vec{x}) \rightarrow g_i(\vec{x})$, with operation symbol g_i corresponding to the function \mathbf{g}_i . There are two cases. If $f_i(\mathbf{nil}, \vec{v})$ is cached, i.e. $(f_i(\mathbf{nil}, \vec{v}), u) \in C$, then $m = 0$ and the case follows trivially. For the case that $f_i(\mathbf{nil}, \vec{v})$ is not cached, by one application of the hypothesis of the outer induction we see

$$\frac{\begin{array}{c} \vdots \\ (C, g_i(\vec{v})) \Downarrow_m (C \cup FC_{\vec{v}}^{\mathbf{g}_i}, u) \end{array} \text{ (IH)}}{(C, f_i(\mathbf{nil}, \vec{v})) \Downarrow_{m+1} (C \cup FC_{\vec{v}}^{\mathbf{g}_i} \cup \{(f_i(\mathbf{nil}, \vec{v}), u)\}, u)} \text{ (update)}$$

where the IH gives additionally $m \leq |FC_{\vec{v}}^{\mathbf{g}_i} \setminus C|$. Combining this with the definition $FC_{\mathbf{nil}, \vec{v}}^{\mathbf{f}_i} = FC_{\vec{v}}^{\mathbf{g}_i} \uplus \{f_i(\mathbf{nil}, \vec{v})\}$ and the assumption that $f_i(\mathbf{nil}, \vec{v})$ is not cached, the case follows.

- For the inductive step, we consider the statement $(C, f_i(\mathbf{bin}(v_1, v_2), \vec{v})) \Downarrow_m (D, u)$. It suffices to consider the case where the call $f_i(\mathbf{bin}(v_1, v_2), \vec{v})$ is not cached. Note that by construction, the program $P_{\mathbf{f}}$ contains a rule

$$f_i(\mathbf{bin}(z, w), \vec{x}) \rightarrow h_i(z, w, \vec{x}, f_1(z, \vec{x}), f_1(w, \vec{x}), f_2(z, \vec{x}), f_2(w, \vec{x})),$$

where the operation symbol f_1 computes the function \mathbf{f}_1 , and so on. Abbreviate the sequence of values $v_1, v_2, \vec{v}, u_1, u_2, u_3, u_4$ by \vec{w} . Furthermore, we introduce following abbreviations for caches that arise during the computation of $f_i(\mathbf{bin}(v_1, v_2), \vec{v})$:

$$C_1 := C \cup FC_{v_1, \vec{v}}^{\mathbf{f}_1} \quad C_2 := C_1 \cup FC_{v_2, \vec{v}}^{\mathbf{f}_1} \quad C_3 := C_2 \cup FC_{v_1, \vec{v}}^{\mathbf{f}_2} \quad C_4 := C_3 \cup FC_{v_2, \vec{v}}^{\mathbf{f}_2} \quad C_5 := C_4 \cup FC_{\vec{w}}^{\mathbf{h}_i}.$$

Using the IH on the recursive calls as well as the IH on \mathbf{h}_i , and tacitly employing Lemma 4.1, we see

$$\frac{\frac{\vdots}{(C, \mathbf{f}_1(v_1, \vec{v})) \Downarrow_{n_1} (C_1, u_1)} \text{ (IH)} \cdots \frac{\frac{\vdots}{(C_3, \mathbf{f}_2(v_2, \vec{v})) \Downarrow_{n_4} (C_4, u_4)} \text{ (IH)} \frac{\frac{\vdots}{(C_4, \mathbf{h}_i(\vec{w})) \Downarrow_{n_5} (C_5, u)} \text{ (IH)}}{(C, \mathbf{h}_i(v_1, v_2, \vec{v}, \mathbf{f}_1(v_1, \vec{v}), \mathbf{f}_1(v_2, \vec{v}), \mathbf{f}_2(v_1, \vec{v}), \mathbf{f}_2(v_2, \vec{v}))) \Downarrow_{\sum_{i=1}^5 n_i} (C_5, u)} \text{ (F-context)}}{(C, f_i(\mathbf{bin}(v_1, v_2), \vec{v})) \Downarrow_{1+\sum_{i=1}^5 n_i} (D, u)} \text{ (update)}$$

where $D = C_5 \cup \{(f_i(\mathbf{bin}(v_1, v_2), \vec{v}), u)\}$. Let $C_0 = C$. Note that the sets $C_i \setminus C_{i-1}$ ($i = 1, \dots, 5$) and C are all pairwise disjoint, and thus we can write $C_5 \setminus C = \bigsqcup_{i=1}^5 (C_i \setminus C_{i-1})$. Since the IH gives $n_i \leq |C_i \setminus C_{i-1}|$, we conclude

$$\sum_{i=1}^5 n_i \leq \sum_{i=1}^5 |C_i \setminus C_{i-1}| = |C_5 \setminus C|.$$

As by definition we have

$$FC_{\mathbf{bin}(v_1, v_2), \vec{v}}^{\mathbf{f}_i} = FC_{v_1, \vec{v}}^{\mathbf{f}_1} \cup FC_{v_2, \vec{v}}^{\mathbf{f}_1} \cup FC_{v_1, \vec{v}}^{\mathbf{f}_2} \cup FC_{v_2, \vec{v}}^{\mathbf{f}_2} \cup FC_{\vec{w}}^{\mathbf{h}_i} \cup \{(f_i, \mathbf{bin}(v_1, v_2), \vec{v})\};$$

$$C_5 = FC_{v_1, \vec{v}}^{\mathbf{f}_1} \cup FC_{v_2, \vec{v}}^{\mathbf{f}_1} \cup FC_{v_1, \vec{v}}^{\mathbf{f}_2} \cup FC_{v_2, \vec{v}}^{\mathbf{f}_2} \cup FC_{\vec{w}}^{\mathbf{h}_i} \cup C,$$

and using the assumption that the call $f_i(\mathbf{bin}(v_1, v_2), \vec{v})$ is not cached in C , we conclude

$$1 + \sum_{i=1}^5 n_i \leq 1 + |C_5 \setminus C| = |FC_{\mathbf{bin}(v_1, v_2), \vec{v}}^{\mathbf{f}_i} \setminus C|. \quad \square$$

The only missing link, then, is bounding the cardinality of $FC_{\vec{v}}^{\mathbf{f}}$ whenever \mathbf{f} is ramified. This is the purpose of the following result:

Lemma 5.3. *If $\mathbf{f} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$, then there is polynomial $p_{\mathbf{f}}$ such that for every \vec{v} , it holds that $|FC_{\vec{v}}^{\mathbf{f}}| \leq p_{\mathbf{f}}(\|\vec{v}\|)$.*

Proof. The proof is by induction on the structure of the proof of $\mathbf{f} \triangleright \mathbf{A} \rightarrow \mathbb{A}_n$. We again consider the most interesting case $\mathbf{f} = \mathbf{f}_1$ with the two ternary functions $\mathbf{f}_1, \mathbf{f}_2 \triangleright \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_0 \rightarrow \mathbb{A}_0$ defined by simultaneous ramified recurrence based on functions $\vec{\mathbf{g}} = \mathbf{g}_1, \mathbf{g}_1$ and $\vec{\mathbf{h}} = \mathbf{h}_1, \mathbf{h}_2$ over binary trees as in (†) on page 4. From the IH, we know that there exist appropriate polynomials $p_{\vec{\mathbf{g}}}$ and $p_{\vec{\mathbf{h}}}$. Please observe that

$$FC_{u, v, w}^{\vec{\mathbf{f}}} = FC_{v, w}^{\vec{\mathbf{g}}} \cup \bigcup_{\mathbf{bin}(y, z) \in \text{STs}(u)} FC_{y, z, v, w, \mathbf{f}_1(y, v, w), \mathbf{f}_1(z, v, w), \mathbf{f}_2(y, v, w), \mathbf{f}_2(z, v, w)}^{\vec{\mathbf{h}}},$$

where expressions like $FC_{\vec{\mathbf{f}}}^{\vec{\mathbf{g}}}$ stand for the union $FC_{\vec{\mathbf{f}}}^{\mathbf{g}_1} \cup FC_{\vec{\mathbf{f}}}^{\mathbf{g}_2}$. This can be easily proved by induction on u , and is a consequence of the way FC is defined. Now, define

$$p_{\vec{\mathbf{f}}}(x) = p_{\vec{\mathbf{g}}}(x) + x \cdot p_{\vec{\mathbf{h}}}(x + 2 \cdot q_{\vec{\mathbf{f}}}(x)),$$

where expressions like $p_{\vec{\mathbf{g}}}$ stand for $p_{\mathbf{g}_1} + p_{\mathbf{g}_2}$ and $q_{\vec{\mathbf{f}}}$ is the polynomial from Lemma 5.1. Let us prove that

this is, indeed, a correct bound:

$$\begin{aligned}
|FC_{u,v,w}^{\vec{f}}| &\leq |FC_{v,w}^{\vec{g}}| + \left| \bigcup_{\mathbf{bin}(y,z) \in \text{STs}(u)} FC_{y,z,v,w,\mathbf{f}_1(y,v,w),\mathbf{f}_1(z,v,w),\mathbf{f}_2(y,v,w),\mathbf{f}_2(z,v,w)}^{\vec{h}} \right| \\
&\leq p_{\vec{g}}(\|v,w\|) + \sum_{\mathbf{bin}(y,z) \in \text{STs}(u)} |FC_{y,z,v,w,\mathbf{f}_1(y,v,w),\mathbf{f}_1(z,v,w),\mathbf{f}_2(y,v,w),\mathbf{f}_2(z,v,w)}^{\vec{h}}| \\
&\leq p_{\vec{g}}(\|v,w\|) + \sum_{\mathbf{bin}(y,z) \in \text{STs}(u)} p_{\vec{h}}(\|y,z,v,w,\mathbf{f}_1(y,v,w),\mathbf{f}_1(z,v,w),\mathbf{f}_2(y,v,w),\mathbf{f}_2(z,v,w)\|) \\
&\leq p_{\vec{g}}(\|v,w\|) + \sum_{\mathbf{bin}(y,z) \in \text{STs}(u)} p_{\vec{h}}(\|u,v,w,\mathbf{f}_1(y,v,w),\mathbf{f}_1(z,v,w),\mathbf{f}_2(y,v,w),\mathbf{f}_2(z,v,w)\|) \\
&\leq p_{\vec{g}}(\|v,w\|) + \sum_{\mathbf{bin}(y,z) \in \text{STs}(u)} p_{\vec{h}}(\|u,v,w\| + \|\mathbf{f}_1(y,v,w),\mathbf{f}_2(y,v,w)\| + \|\mathbf{f}_1(z,v,w),\mathbf{f}_2(z,v,w)\|) \\
&\leq p_{\vec{g}}(\|v,w\|) + \sum_{\mathbf{bin}(y,z) \in \text{STs}(u)} p_{\vec{h}}(\|u,v,w\| + q_{\vec{f}}(\|y,v,w\|) + q_{\vec{f}}(\|z,v,w\|)) \\
&\leq p_{\vec{g}}(\|u,v,w\|) + \sum_{\mathbf{bin}(y,z) \in \text{STs}(u)} p_{\vec{h}}(\|u,v,w\| + q_{\vec{f}}(\|u,v,w\|) + q_{\vec{f}}(\|u,v,w\|)) \\
&\leq p_{\vec{g}}(\|u,v,w\|) + \|u\| \cdot p_{\vec{h}}(\|u,v,w\| + 2 \cdot q_{\vec{f}}(\|u,v,w\|)) \\
&\leq p_{\vec{g}}(\|u,v,w\|) + \|u,v,w\| \cdot p_{\vec{h}}(\|u,v,w\| + 2 \cdot q_{\vec{f}}(\|u,v,w\|)) \\
&= p_{\vec{f}}(\|u,v,w\|).
\end{aligned}$$

This concludes the proof. \square

The following, then, is just a corollary of Lemma 5.2 and Lemma 5.3.

Lemma 5.4. *If $\mathbf{f} \triangleright \mathbb{A} \rightarrow \mathbb{A}_n$, then there is a polynomial $p_{\mathbf{f}} : \mathbb{N} \rightarrow \mathbb{N}$ such that for every \vec{v} , $(\emptyset, \mathbf{f}(\vec{v})) \Downarrow_m (C, u)$, with $m \leq p_{\mathbf{f}}(\|\vec{v}\|)$.*

Using this final lemma and invariance (Theorem 4.15) we arrive at our main result.

Theorem 5.5. *Let $\mathbf{f} : \mathbb{A}_{p_1} \times \dots \times \mathbb{A}_{p_k} \rightarrow \mathbb{A}_m$ be a function defined by general ramified simultaneous recursion. There exists then a polynomial $p_{\mathbf{f}} : \mathbb{N}^k \rightarrow \mathbb{N}$ such that for all inputs v_1, \dots, v_k , a DAG representation of $\mathbf{f}(v_1, \dots, v_k)$ is computable in time $p_{\mathbf{f}}(|v_1|, \dots, |v_k|)$.*

Example 5.6 (Continued from Example 4.16). In Example 3.4 we already indicated that the function `rabbits` : $\mathbb{N} \rightarrow \mathbb{F}$ from Section 2 is definable by GRSR. As a consequence of Theorem 5.5, 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 [22–24] and term rewrite systems [21, 25] could be proved. Several works (e.g. [18, 19, 26]) 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] D. Leivant, Ramified Recurrence and Computational Complexity I: Word Recurrence and Poly-time, in: Feasible Mathematics II, Progress in Computer Science and Applied Logic, vol. 13, Birkhäuser Science, 320–343, 1995.
- [2] U. Dal Lago, S. Martini, M. Zorzi, General Ramified Recurrence is Sound for Polynomial Time, in: Proc. of 1st International Workshop on Developments in Implicit Complexity, vol. 23 of *Electronic Proceedings in Theoretical Computer Science*, 47–62, 2010.
- [3] P. Van Emde Boas, Machine Models and Simulation, in: Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), The MIT Press, 1–66, 1990.
- [4] D. Leivant, Stratified Functional Programs and Computational Complexity, in: Proc. of 20th Annual Symposium on Principles of Programming Languages, Association for Computing Machinery, 325–333, 1993.
- [5] S. Bellantoni, Predicative Recursion and Computational Complexity, Ph.D. thesis, University of Toronto, 1992.
- [6] S. Bellantoni, Predicative Recursion and the Polytime Hierarchy, in: Feasible Mathematics II, Progress in Computer Science and Applied Logic, Birkhäuser Science, 15–29, 1994.
- [7] I. Oitavem, Implicit Characterizations of Pspace., in: Proof Theory in Computer Science, 170–190, 2001.
- [8] T. Arai, N. Eguchi, A New Function Algebra of EXPTIME Functions by Safe Nested Recursion, ACM Transactions on Computational Logic 10 (4).
- [9] G. Bonfante, R. Kahle, J.-Y. Marion, I. Oitavem, Recursion Schemata for NCK, in: Proc. of 22nd European Association for Computer Science Logic, vol. 5213 of *Lecture Notes in Computer Science*, Springer Verlag Heidelberg, 49–63, 2008.
- [10] N. Danner, J. S. Royer, Ramified Structural Recursion and Corecursion, CoRR abs/1201.4567.
- [11] B. Hoffmann, Term Rewriting with Sharing and Memoization, in: Proc. of 3rd Algebraic and Logic Programming, vol. 632 of *Lecture Notes in Computer Science*, Springer Verlag Heidelberg, 128–142, 1992.
- [12] M. Avanzini, U. Dal Lago, On Sharing, Memoization, and Polynomial Time, in: Proc. of 32nd International Symposium on Theoretical Aspects of Computer Science, vol. 30 of *Leibniz International Proceedings in Informatics*, Leibniz-Zentrum für Informatik, 62–75, 2015.
- [13] S. Bellantoni, S. Cook, A new Recursion-Theoretic Characterization of the Polytime Functions, Computational Complexity 2 (2) (1992) 97–110.
- [14] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
- [15] E. A. Cichon, A. Weiermann, Term Rewriting Theory for the Primitive Recursive Functions, Annals of Pure and Applied Logic 83 (3) (1997) 199–223.
- [16] D. Plump, Essentials of Term Graph Rewriting, Electronic Notes in Theoretical Computer Science 51 (2001) 277–289.
- [17] H. P. Barendregt, M. v. Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, M. R. Sleep, Term Graph Rewriting, in: Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe, vol. 259 of *Lecture Notes in Computer Science*, Springer Verlag Heidelberg, 141–158, 1987.
- [18] J.-Y. Marion, Analysing the Implicit Complexity of Programs, Information and Computation 183 (2003) 2–18.
- [19] G. Bonfante, J.-Y. Marion, J.-Y. Moyon, Quasi-interpretations: A Way to Control Resources, Theoretical Computer Science 412 (25) (2011) 2776–2796.
- [20] N. Hirokawa, G. Moser, Automated Complexity Analysis Based on the Dependency Pair Method, in: Proc. of 4th International Joint Conference on Automated Reasoning, vol. 5195 of *Lecture Notes in Artificial Intelligence*, Springer Verlag Heidelberg, 364–380, 2008.
- [21] U. Dal Lago, S. Martini, Derivational Complexity is an Invariant Cost Model, in: Proc. of 1st International Workshop on Foundational and Practical Aspects of Resource Analysis, vol. 6324 of *Lecture Notes in Computer Science*, Springer Verlag Heidelberg, 100–113, 2009.
- [22] B. Accattoli, U. Dal Lago, On the Invariance of the Unitary Cost Model for Head Reduction, in: Proc. of 23rd International Conference on Rewriting Techniques and Applications, vol. 15 of *Leibniz International Proceedings in Informatics*, Leibniz-Zentrum für Informatik, 22–37, 2012.
- [23] U. Dal Lago, S. Martini, On Constructor Rewrite Systems and the Lambda Calculus, Logical Methods in Computer Science 8 (3) (2012) 1–27.
- [24] B. Accattoli, U. Dal Lago, Beta Reduction is Invariant, Indeed, in: Proc. of 23rd European Association for Computer Science Logic, Association for Computing Machinery, 8:1–8:10, 2014.
- [25] M. Avanzini, G. Moser, Closing the Gap Between Runtime Complexity and Polytime Computability, in: Proc. of 21st International Conference on Rewriting Techniques and Applications, vol. 6 of *Leibniz International Proceedings in Informatics*, Leibniz-Zentrum für Informatik, 33–48, 2010.
- [26] P. Baillot, U. Dal Lago, J. Moyon, On Quasi-interpretations, Blind Abstractions and Implicit Complexity, Mathematical Structures in Computer Science 22 (4) (2012) 549–580.