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 *string* 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 strings, lists, but also trees (see Section 3). We then extend the underlying rewriting based semantics with *memoization*, i.e. intermediate results are automatically tabulated to avoid expensive re-computation (Section 4). As standard for functional programming languages such as Haskell or OCaml, data is stored in a *heap*, facilitating *sharing* of common sub-expression. To measure the *runtime* of such programs, we employ a novel cost model, called *memoized runtime complexity*, where each function application counts one time unit, but lookups of tabulated calls do not have to be accounted.

2. Our *invariance theorem* (see Theorem 4.14) relates, within a polynomial overhead, the memoized runtime complexity of programs to the cost of implementing the defined functions on a classical model of computation, e.g. *Turing* or *random access machines*. The invariance theorem thus confirms that our cost model truthfully represents the computational complexity of the defined function.

3. We extend upon Leivant's notion of *ramified recursive functions* [3] 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 strings* is sound for polynomial time, even in presence of simultaneous recursive definitions, is known for a long time [4]. Variations of predicative recursion have been later considered and proved to characterize classes like PH [5], PSpace [6], ExpTime [7] or NC [8]. Predicative recursion on trees has been claimed to be sound for polynomial time in the original paper by Leivant [3], the long version of which only deals with strings [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 [9]; 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 [10], 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 [11].

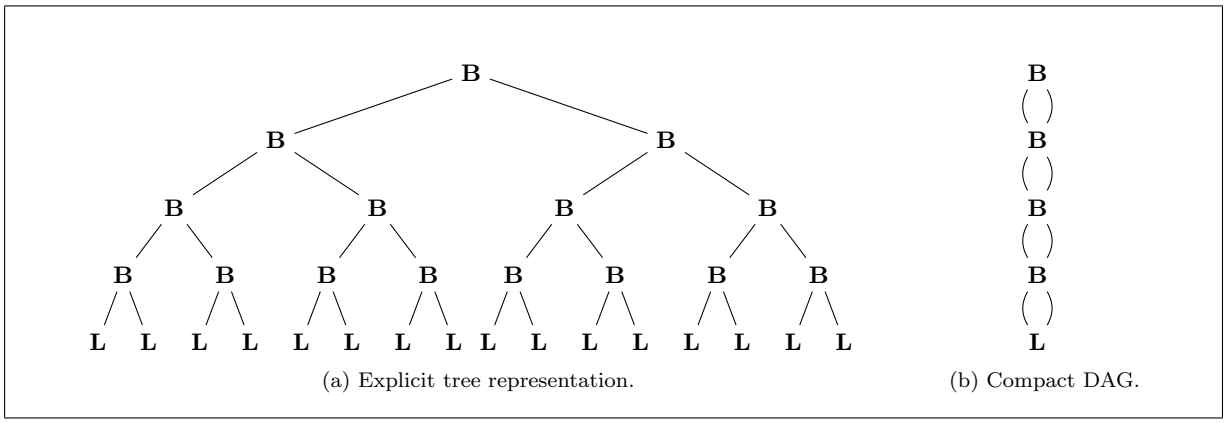
2. The Need for Sharing and Memoization

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.

In Leivant's system [1], functions and variables are equipped with a *tier*. Composition must preserve tiers and, crucially, in a function defined by primitive recursion the tier of the recurrence parameter must be higher than the tier of the recursive call. This form of *ramification* of functions effectively tames primitive recursion, resulting in a characterisation of the class of *polytime computable functions*.

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

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



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

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

The function `tree` is defined by primitive recursion, essentially from basic functions. As a consequence, it is easily seen to be ramified in the sense of Leivant. Even though the number of recursive steps is linear in the input, the result of $\mathtt{tree}(\mathbf{S}^n(\mathbf{0}))$ 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 `tree` a polytime function. The way out is *sharing*: the complete binary tree of height n can be compactly represented as a *directed acyclic graph* (*DAG* for short) of linear size (see Figure 1). Indeed, using the compact DAG representation it is easy to see that the function `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. The following example computes the genealogical tree associated with *Fibonacci's rabbit problem* for $n \in \mathbb{N}$ generations. Rabbits come in pairs. After one generation, each *baby* rabbit pair ($\mathbf{N}$) matures. In each generation, an *adult* rabbit pair ($\mathbf{M}$) bears one pair of babies.

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

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

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

3. Preliminaries

3.1. General Ramified Simultaneous Recurrence

Let $\mathbb{A}$ denote a *(finite and untyped) signatures* of *constructors* $\mathbf{c}_1, \ldots, \mathbf{c}_k$, each equipped with an arity $\mathsf{ar}(\mathbf{c}_i) \in \mathbb{N}$. In the following, the set of terms $\mathcal{T}(\mathbb{A})$ over the signature $\mathbb{A}$, defined as usual, is also denoted by

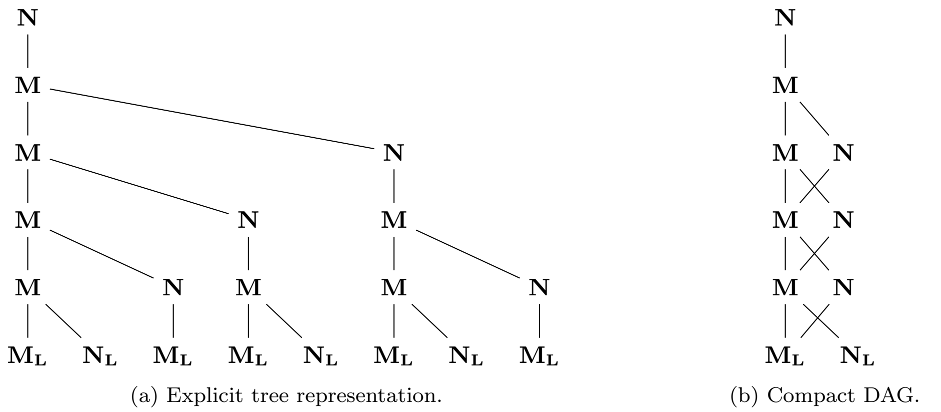


Figure 2: Genealogical rabbit tree up to the sixth generation, as computed by $\mathtt{rabbits}(\mathbf{S}^6(\mathbf{0}))$.

$\mathbb{A}$ if this does not create ambiguities. We are interested in total functions from $\mathbb{A}^n = \underbrace{\mathbb{A} \times \ldots \times \mathbb{A}}_{n \text{ times}}$ to $\mathbb{A}$.

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

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

$$\mathtt{f}_{\mathbf{c}}(x_1, \ldots, x_{\mathsf{ar}(\mathbf{c})}) = \mathbf{c}(x_1, \ldots, x_{\mathsf{ar}(\mathbf{c})}) \ .$$

- For each $1 \le n \le m$, the (m,n)-*projection function* $\Pi^m_n : \mathbb{A}^m \to \mathbb{A}$ defined as follows:

$$\Pi^m_n(x_1 \ldots, x_m) = x_n \ .$$

Definition 3.2. We define the class $\textsc{SimRec}(\mathbb{A})$ 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 $\mathtt{f} : \mathbb{A}^n \to \mathbb{A}$ and n functions $\mathtt{g}_1, \ldots, \mathtt{g}_n$, all of them from $\mathbb{A}^m$ to $\mathbb{A}$, the *composition* $\mathtt{h} = \mathtt{f} \circ (\mathtt{g}_1, \ldots, \mathtt{g}_n)$ is a function from $\mathbb{A}^m$ to $\mathbb{A}$ defined as follows:

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

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

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

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

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

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

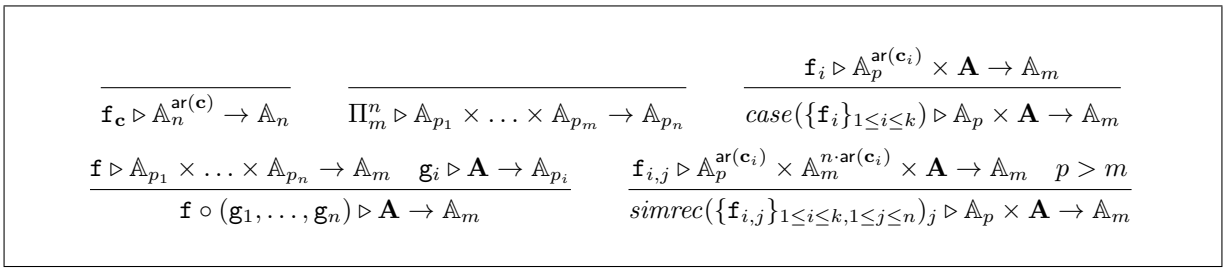


Figure 3: Tiering as a Formal System.

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

Definition 3.3. We call a function $\mathtt{f} \in \text{SimRec}(\mathbb{A})$ definable by *general ramified simultaneous recurrence* (*GRSR* for short) if $\mathtt{f} \triangleright \mathbb{A}_{p_1} \times \ldots \times \mathbb{A}_{p_{\mathsf{ar}(\mathtt{f})}} \to \mathbb{A}_m$ holds.

Remark 2. Consider the *word algebra* $\mathbb{W} = \{\epsilon, \mathbf{a}, \mathbf{b}\}$ consisting of a constant ϵ and two unary constructors $\mathbf{a}$ and $\mathbf{b}$, which is in bijective correspondence to the set of binary words. Then the functions definable by ramified simultaneous recurrence over $\mathbb{W}$ includes the ramified recursive functions from Leivant [1], and consequently all polytime computable functions.

Example 3.4.

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

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

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

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

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

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

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

$$\frac{\mathtt{f} \in \mathcal{F} \quad t_i \downarrow v_i \quad f(v_1, \ldots, v_k) \downarrow v}{f(t_1, \ldots, t_k) \downarrow v}\,(\mathcal{F}\text{-}\mathtt{context})$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \ldots, t_k) \downarrow \mathbf{c}(v_1, \ldots, v_k)}\,(\mathcal{C}\text{-}\mathtt{context})$$

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

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

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

3.2. Computational Model, Syntax and Semantics

We assume modest familiarity with rewriting, see the book of Baader and Nipkow [12] 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* $\mathsf{STs}(t)$ of a term t is defined by $\mathsf{STs}(t) := \{t\}$ if $t \in \mathcal{V}$ and $\mathsf{STs}(t) := \bigcup_{1 \le i \le \mathsf{ar}(f)} \mathsf{STs}(t_i) \cup \{t\}$ if $t = f(t_1, \ldots, t_{\mathsf{ar}(f)})$. A *substitution*, is a finite mapping σ from variables to terms. By $t\sigma$ we denote the term obtained by replacing in t all variables x in the domain of σ by $\sigma(x)$.

Definition 3.5. A *program* $\mathtt{P}$ is given as a triple $(\mathcal{F}, \mathcal{C}, \mathcal{R})$ consisting of two disjoint signatures $\mathcal{F}$ and $\mathcal{C}$ of *operation symbols* $f_1, \ldots, f_m$ and *constructors* $\mathbf{c}_1, \ldots, \mathbf{c}_n$ respectively, and a *finite* set $\mathcal{R}$ of *rules* $l \to r$ over terms $l, r \in \mathcal{T}(\mathcal{F} \cup \mathcal{C}, \mathcal{V})$, satisfying:

- the *left-hand side* l is of the form $f(p_1, \ldots, p_k)$ with $f \in \mathcal{F}$ and all arguments p_j are *patterns*, i.e. terms formed from the constructors $\mathcal{C}$ 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 *ortoghonal*, that is, the following two requirements are met:

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

We keep the program $\mathtt{P} = (\mathcal{F}, \mathcal{C}, \mathcal{R})$ 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. [12]. 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 $v, u, \ldots$ 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 $\mathtt{P}$ computes the function $\mathtt{f} : \mathcal{T}(\mathcal{C})^k \to \mathcal{T}(\mathcal{C})$ if there exists an operation $f \in \mathcal{F}$ such that $\mathtt{f}(v_1, \ldots, v_k) = v$ if and only if $f(v_1, \ldots, v_k) \downarrow v$ holds for all inputs $v_i \in \mathcal{T}(\mathcal{C})$.

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

- the set of operation symbols underlying $\mathtt{P}_{\mathtt{rabbits}}$ consists of (i) ternary symbols P_2^3 and P_3^3 implementing the projections Π_2^3 and Π_3^3; (ii) constant symbols $f_{\mathbf{M_L}}, f_{\mathbf{N_L}}$, a binary symbol $f_{\mathbf{M}}$ and a unary symbol $f_{\mathbf{N}}$ implementing the constructor functions associated with $\mathbb{T}$; (iii) two operation symbols $\mathit{comp}[f_{\mathbf{M}}; P_2^3, P_3^3]$ and $\mathit{comp}[f_{\mathbf{N}}; P_3^3](x_1, x_2, x_3)$ implementing the composition of the mentioned operations; (iv) two symbols

$$\mathtt{a} := \mathit{simrec}[[f_{\mathbf{M_L}}, \mathit{comp}[f_{\mathbf{M}}; P_2^3, P_3^3]], [f_{\mathbf{N_L}}, \mathit{comp}[f_{\mathbf{N}}; P_3^3]]_1 \ ;$$
$$\mathtt{b} := \mathit{simrec}[[f_{\mathbf{M_L}}, \mathit{comp}[f_{\mathbf{M}}; P_2^3, P_3^3]], [f_{\mathbf{N_L}}, \mathit{comp}[f_{\mathbf{N}}; P_3^3]]_2 \ ,$$

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

- the set of constructors underlying $\mathtt{P}_{\mathtt{rabbits}}$ consists of the constructors from $\mathbb{T} \cup \mathbb{N}$;

- the set of rules underlying $\mathtt{P}_{\mathtt{rabbits}}$ is given as follows, where $\vec{x} := x_1, x_2, x_3$.

$$\begin{array}{rrr}
P_2^3(\vec{x}) \to x_2 & f_{\mathbf{M_L}} \to \mathbf{M_L} & f_{\mathbf{M}}(x_1, x_2) \to \mathbf{M}(x_1, x_2) \\
P_3^3(\vec{x}) \to x_3 & f_{\mathbf{N_L}} \to \mathbf{N_L} & f_{\mathbf{N}}(x_1) \to \mathbf{N_L}(x_1) \\
\mathtt{a}(\mathbf{0}) \to f_{\mathbf{M_L}} & \mathtt{a}(\mathbf{S}(n)) \to \mathit{comp}[f_{\mathbf{M}}; P_2^3, P_3^3](n, \mathtt{a}(n), \mathtt{b}(n)) & \mathit{comp}[f_{\mathbf{M}}; P_2^3, P_3^3](\vec{x}) \to f_{\mathbf{M}}(P_2^3(\vec{x}), P_3^3(\vec{x})) \\
\mathtt{b}(\mathbf{0}) \to f_{\mathbf{N_L}} & \mathtt{b}(\mathbf{S}(n)) \to \mathit{comp}[f_{\mathbf{N}}; P_3^3](n, \mathtt{a}(n), \mathtt{b}(n)) & \mathit{comp}[f_{\mathbf{N}}; P_3^3](\vec{x}) \to f_{\mathbf{N}}(P_3^3(\vec{x})) \\
r(\mathbf{0}) \to f_{\mathbf{N_L}} & r(\mathbf{S}(n)) \to \mathtt{b}(n) \ . &
\end{array}$$

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

The construction is straight-forward to generalize to arbitrary functions in SimRec($\mathbb{A}$), see e.g. the work on term rewriting characterizations by Cichon and Weiermann [13] for a more formal treatment.

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

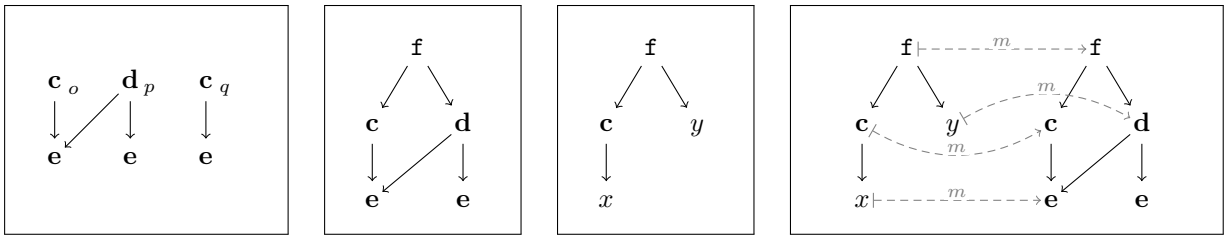
- the set of operations $\mathcal{F}_{\mathtt{f}}$ contains for each function `g` underlying the definition of `f` a fresh operation symbol g;
- the set of constructors $\mathcal{C}_{\mathtt{f}}$ contains the constructors of $\mathbb{A}$;
- the set of rules $\mathcal{R}_{\mathtt{f}}$ contains for each equation $l = r$ underlying the definition of a function `g` associated with $g \in \mathcal{F}_{\mathtt{f}}$ the *orientation* $\langle l \rangle \to \langle r \rangle$, 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 SimRec($\mathbb{A}$), the program $\mathtt{P}_{\mathtt{f}}$ is finite.

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

3.3. Term Graphs

We borrow key concepts from *term graph rewriting* (see e.g. the survey of Plump [14] for an overview) and follow the presentation of Barendregt et al. [15]. A *term graph* T over a signature $\mathcal{F}$ is a *directed acyclic graph* whose nodes are labeled by symbols in $\mathcal{F} \cup \mathcal{V}$, and where outgoing edges are ordered. Formally, T is a triple $(N, \mathsf{suc}, \mathsf{lab})$ consisting of *nodes* N, a *successors function* $\mathsf{suc} : N \to N^*$ and a *labeling function* $\mathsf{lab} : N \to \mathcal{F} \cup \mathcal{V}$. We require that term graphs are *compatible* with $\mathcal{F}$, in the sense that for each node $o \in N$, if $\mathsf{lab}_T(o) = f \in \mathcal{F}$ then $\mathsf{suc}_T(o) = [o_1, \ldots, o_{\mathsf{ar}(f)}]$ and otherwise, if $\mathsf{lab}_T(o) = x \in \mathcal{V}$, $\mathsf{suc}_T(o) = [\,]$. In the former case, we also write $T(o) = f(o_1, \ldots, o_{\mathsf{ar}(f)})$, the latter case is denoted by $T(o) = x$. We define the *successor relation* $\rightharpoonup_T$ on nodes in T such that $o \rightharpoonup_T p$ holds iff p occurs in $\mathsf{suc}(o)$, if p occurs at the i^{th}



(a) A term graph T such that $[o]_T = [q]_T = \mathbf{c}(\mathbf{e})$ and $[p]_T = \mathbf{d}(\mathbf{e}, \mathbf{e})$.

(b) The rooted term graph $\mathtt{f}(T\lfloor o, T\lfloor p)$.

(c) The rooted term graph $\triangle(\mathtt{f}(\mathbf{c}(x), y))$.

(d) A matching morphism $\triangle(\mathtt{f}(\mathbf{c}(x), y)) \geqslant_m \mathtt{f}(T\lfloor o, T\lfloor p)$.

Figure 5: Term graphs and morphisms.

position we also write $o \xrightarrow{i}_T p$. Throughout the following, we consider only *acyclic* term graphs, that is, when $\rightharpoonup_T$ is acyclic. Hence the *unfolding* $[o]_T$ of T *at* node o, defined as follows, results in a finite term.

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

We called the term graph T *rooted* if there exists a unique node o, the *root* of T, with $o \rightharpoonup_T^* p$ for every $p \in N$. We denote by $T\lfloor o$ the *sub-graph* of T *rooted* at o. For a symbol $f \in \mathcal{F}$ and nodes $\{o_1, \ldots, o_{\mathsf{ar}(f)}\} \subseteq N$, we write $f(T\lfloor o_1, \ldots, T\lfloor o_{\mathsf{ar}(f)})$ for the term graph $S\lfloor o_f$, where S is defined as the extension of T by a fresh node $o_f \notin N$ with $S(o_f) = f(o_1, \ldots, o_{\mathsf{ar}(f)})$. Every (linear) term t is trivially representable as a *canonical tree* $\triangle(t)$ unfolding to t, using a fresh node for each occurrence of a subterm in t. Compare Figure 5a, b and, c, which illustrate key concepts.

For two rooted term graphs $T = (N_T, \mathsf{suc}_T, \mathsf{lab}_T)$ and $S = (N_S, \mathsf{suc}_S, \mathsf{lab}_S)$, a mapping $m : N_T \to N_S$ is called *morphic* in $o \in N_T$ if (i) $\mathsf{lab}_T(o) = \mathsf{lab}_S(m(o))$ and (ii) $o \xrightarrow{i}_T p$ implies $m(o) \xrightarrow{i}_S m(p)$ for all appropriate i. A *homomorphism* from T to S is a mapping $m : N_T \to N_S$ that (i) maps the root of T to the root of S and that (ii) is morphic in all nodes $o \in N_T$ not labeled by a variable. We write $T \geqslant_m S$ to indicate that m is, possibly an extension of, a homomorphism from T to S. See also Figure 5d for an example.

The following proposition relates matching on terms and homomorphisms on trees, which 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 $\triangle(t)$ labeled by x, which we denote by o_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 $\triangle(t) \geqslant_m T\lfloor o$ then there exists a substitution σ such that $t\sigma = [o]_T$.*
2. *If $t\sigma = [o]_T$ holds for some substitution σ then there exists a homomorphism $\triangle(t) \geqslant_m T\lfloor o$.*

Here, the substitution σ and homomorphism m satisfy $\sigma(x) = [m(o_x)]_T$ 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 $\triangle(t) \geqslant_m S\lfloor o$. When t is a variable, the substitution $\sigma := \{t \mapsto [o]_S\}$ satisfies $t\sigma = [o]_S$. Since $m(\epsilon) = o$ by definition, we conclude the base case. For the inductive step, assume $t = f(t_1, \ldots, t_k)$. For $i = 1, \ldots, k$, 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 $\triangle(t_i)$ one verifies $\triangle(t_i) \geqslant_{m_i} [n_i]_S$. Thus the induction hypothesis (IH) yields substitutions σ_i for $i = 1, \ldots, k$, such that $t_i\sigma_i = [n_i]_S$. Without loss of generality, suppose σ_i is restricted to variables in t_i. Define $\sigma := \bigcup_{i=1}^{k} \sigma_i$. Then $t\sigma = f(t_1\sigma_1, \ldots, t_k\sigma_k) = f([o_1]_S, \ldots, [o_k]_S) = [o]_S$, where the last equality follows as m is morphic in o. Moreover, from the shape of σ_i and m_i it is not difficult to see that by construction the substitution σ and homomorphism m are related as claimed by the lemma.

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

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

$$\frac{(f(v_1, \ldots, v_k), v) \in C}{(C, f(v_1, \ldots, v_k)) \Downarrow_0 (C, v)} (\texttt{read})$$

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

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

For the direction (2), suppose $t\sigma = [o]_T$. If t is a variable and thus $\triangle(t)$ consists of a single node labeled by t, trivially $\triangle(t) \geqslant_m T\downarrow o$ holds for m the homomorphism which maps the root of $\triangle(t)$ to o. Observe that $\sigma(t) = [o]_T = [m(\epsilon)]_T$, which concludes the base case. For the inductive step suppose $t = f(t_1, \ldots, t_k)$, hence $T(o) = f(o_1, \ldots, o_k)$, $t_i\sigma = [o_i]_T$ $(i = 1, \ldots, k)$ and thus by IH $\triangle(t_i) \geqslant_{m_i} [o_i]_T$ for homomorphisms m_i. Define the function m by $m(\epsilon) := o$ and $m(i{\cdot}p) := m_i(p)$ for all $i = 1, \ldots, k$ and positions $i{\cdot}p$ of t. Observe that m is defined on all nodes of $\triangle(t)$. By definition of m one finally concludes the lemma, using the IH together with the equalities $\triangle(t)(i{\cdot}p) = \triangle(t_i)(p)$ for positions $i{\cdot}p$ $(i = 1, \ldots, k)$ of $\triangle(t)$. □

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. [16, 17]. 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 $\mathtt{P} = (\mathcal{F}, \mathcal{C}, \mathcal{R})$. 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 $(f(v_1, \ldots, v_{\mathsf{ar}(f)}), u)$, with $f \in \mathcal{F}$ and $v_1, \ldots, v_{\mathsf{ar}(f)}$ as well as u are values. We require that caches C are *proper* with respect to the program $\mathtt{P}$, i.e. C associates function calls $f(v_1, \ldots, v_k)$ with their corresponding result u. Formally, a cache C is called proper if $(f(v_1, \ldots, v_k), u) \in C$ implies $f(v_1, \ldots, v_k) \downarrow u$.

Figure 6 collects the *memoizing operational semantics* of the program $\mathtt{P}$. 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 (`apply`) from Figure 4 is split into two rules (`read`) and (`update`). The former performs a read from the cache. The latter performs the reduction in case the corresponding function call is not *tabulated*, updating the cache after the call has been completely evaluated. 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.

1. Suppose that the last rule in Π has the form

$$\frac{t_i \downarrow v_i \quad \mathsf{f}(p_1, \ldots, p_k) \to r \in \mathcal{R} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\mathsf{f}(t_1, \ldots, t_k) \downarrow v}\,(\mathcal{F}\texttt{-context})$$

   By induction hypothesis, we obtain proper caches $D_0, \ldots, D_k$ 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 $(D_k, \mathsf{f}(v_1, \ldots, v_k)) \Downarrow_n (D, v)$ for D a proper cache. We distinguish two cases. Consider the case $(\mathsf{f}(v_1, \ldots, v_k), u) \in D_k$ for some u. Using that $\mathsf{f}(v_1, \ldots, v_k) \downarrow u$ implies $v = u$ for orthogonal programs, we conclude the case by one application of rule (`read`). Otherwise, we conclude by rule (`update`) using the induction hypothesis on $r\sigma \downarrow v$. Note that the resulting cache is also in this case proper.

2. The final case follows directly from induction hypothesis, using the rule ($\mathcal{C}$-`context`).

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

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

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

   Observe that as in the previous direction, all involved caches are proper. Thus by induction hypothesis, we see that $t_i \downarrow v_i$ and $\mathsf{f}(v_1, \ldots, v_k) \downarrow v$, holds and conclude by one application of rule ($\mathcal{F}$-`context`).

2. 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 induction hypothesis.

□

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:

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

Here, $\ell_1, \ldots, \ell_k, \ell \in \mathsf{Loc}$, $\mathsf{f} \in \mathcal{F}$ 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 $\langle \mathsf{f}(\ell_1, \ldots, \ell_k), e\rangle$ indicates that the partially evaluated expression e descends from a call $\mathsf{f}(v_1, \ldots, v_k)$, with arguments v_i stored at location ℓ_i on the heap. A context E is an expression with a unique *hole*, denoted as $\Box$, where all sub-expression to the left of the hole are references pointing to values. This syntactic restriction is used to implement a *left-to-right*, *call-by-value* evaluation order. We denote by $E[e]$ the expression obtained by replacing the hole in E by e.

$$\frac{(f(\ell_1,\dots,\ell_k),\ell) \notin D \quad f(p_1,\dots,p_k) \to r \in \mathcal{R} \quad \triangle(f(p_1,\dots,p_k)) \gg_m f(H\lfloor\ell_1,\dots,H\lfloor\ell_k) \quad \sigma_m := \{x \mapsto m(\ell_x) \mid \ell_x \in \mathsf{Loc},\ T(\ell_x) = x \in \mathcal{V}\}}{(D,H,E[f(\ell_1,\dots,\ell_k)]) \to_{\mathtt{R}} (D,H,E[\langle f(\ell_1,\dots,\ell_k), r\sigma_m\rangle])}\,(\mathtt{apply})$$

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

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

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

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

A *configuration* is a triple (D,H,e) consisting of a *cache* D, *heap* H and expression e. Unlike before, the cache D consists of pairs of the form $(f(\ell_1,\dots,\ell_k),\ell)$ where, instead of values, we store references $\ell_1,\dots,\ell_k,\ell$ pointing to the heap. The heap H is represented as a (multi-rooted) term graph H with nodes in Loc and constructors $\mathcal{C}$ as labels. If ℓ is a node of H, then we say that H stores at location ℓ the value $[\ell]_H$ obtained by unfolding H starting from location ℓ. We keep the heap in a *maximally shared* form, that is, $H(\ell_a) = \mathbf{c}(\ell_1,\dots,\ell_k) = H(\ell_b)$ implies $\ell_a = \ell_b$ for two locations ℓ_a,ℓ_b of H. The operation $\mathsf{merge}(H,\mathbf{c}(\ell_1,\dots,\ell_k))$, defined as follows, is used to extend the heap H with a constructor $\mathbf{c}$ whose arguments point to $\ell_1,\dots,\ell_k$, retaining maximal sharing. For $\ell_1,\dots,\ell_k \in N$ we define

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

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

Figure 7 defines small-step semantics for the program $\mathtt{P}$ as the composition of four relations $\to_{\mathtt{R}}$, $\to_{\mathtt{r}}$, $\to_{\mathtt{s}}$ and $\to_{\mathtt{m}}$, which are defined by the rules (`apply`), (`read`), (`store`) and (`merge`), respectively. The relation $\to_{\mathtt{R}}$, 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 $\to_{\mathtt{s}}$ removes this marker and adds a corresponding entry to the heap. Finally, the relation $\to_{\mathtt{r}}$ implements memoization as in the operational semantics, the relation $\to_{\mathtt{m}}$ 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 $\to_{\mathtt{rsm}}$ the relation $\to_{\mathtt{r}} \cup \to_{\mathtt{s}} \cup \to_{\mathtt{m}}$, likewise we abbreviate $\to_{\mathtt{R}} \cup \to_{\mathtt{rsm}}$ by $\to_{\mathtt{Rrsm}}$. Furthermore, we define $\to_{\mathtt{R/rsm}} := \to^*_{\mathtt{rsm}} \cdot \to_{\mathtt{R}} \cdot \to^*_{\mathtt{rsm}}$. Hence the *m-fold composition* $\to^m_{\mathtt{R/rsm}}$ corresponds to a $\to_{\mathtt{Rrsm}}$-reduction with precisely m applications of $\to_{\mathtt{R}}$. 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* $(D_1, H_1, e_1) \to_{\mathtt{Rrsm}} (D_2, H_2, e_2)$ *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 $(D_1, H_1, e_1) \to_{\mathtt{Rrsm}} (D_2, H_2, e_2)$. We check that (D_2, H_2, e_2) is well-formed by case analysis on $\to_{\mathtt{Rrsm}}$.

1. *The heap* H_2 *is maximally shared*: As only the relation $\to_{\mathtt{m}}$ modifies the heap, it suffices to consider the case $(D_1, H_1, e_1) \to_{\mathtt{m}} (D_2, H_2, e_2)$. Then $(H_2, \ell) = \mathsf{merge}(H_1, \mathbf{c}(\ell_1, \dots, \ell_k))$ 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 $\to_{\mathtt{s}}$ 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 $\to_{\mathtt{R}}$ and $\to_{\mathtt{s}}$ 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 $\to_{\mathtt{m}}$ 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 $\langle \mathsf{f}(\ell_1, \dots, \ell_k), \cdot \rangle$. Formally, we define

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

Likewise, we set $[D]_H := \big\{([e]_H, [\ell]_H) \mid (e, \ell) \in D\big\}$. 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 $\to_H^*$-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* $(D_1, H_1, e_1) \to_{\mathtt{Rrsm}} (D_2, H_2, e_2)$ *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

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

with $(H_2, \ell) = \mathsf{merge}(H_1, \mathbf{c}(\ell_1, \dots, \ell_k))$. 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 follows essentially by definition of $\to_{\mathtt{m}}$.

Lemma 4.6. *Let $(D, H, E[v])$ be a configuration for a value v. Then $(D, H, E[v]) \to_{\mathtt{m}}^* (D, H', E[\ell])$ 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 $(D, H, E[v]) \to_{\mathtt{m}} (D, H', E[E'[\ell]])$, where $(H', \ell) = \mathsf{merge}(H, \mathbf{c}(\ell_1, \dots, \ell_k))$. 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 induction hypothesis. □

An *initial configuration* is a well-formed configuration of the form $(\varnothing, H, e)$ with H a maximally shared heap and $e = \mathit{f}(v_1, \dots, v_k)$ 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 4.7 (Simulation)**.** *Let $(\varnothing, H, e)$ be an initial configuration. If $(\varnothing, [e]_H) \Downarrow_m (C, v)$ holds for $m \geq 1$ then there exists a cache D, heap G and location ℓ in G such that $(\varnothing, H, e) \to_{\mathtt{R/rsm}}^m (D, G, \ell)$ 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 $\langle \mathit{f}(v_1, \dots, v_k), e' \rangle$. We show the following claim:

Claim. For every proper configuration (D_1, H_1, e_1), $([D_1]_{H_1}, [e_1]_{H_1}) \Downarrow_m (C_2, v)$ implies $(D_1, H_1, e_1) \to_{\mathtt{rsm}}^* \cdot \to_{\mathtt{R/rsm}}^m (D_2, H_2, \ell)$ with $([D_2]_{H_2}, [\ell]_{H_2}) = (C_2, v)$.

Observe that $\to_{\mathtt{rsm}}^* \cdot \to_{\mathtt{R/rsm}}^m = \to_{\mathtt{R/rsm}}^m$ whenever $m > 0$. Since an initial configuration is trivially proper, the lemma follows from the claim. To prove the claim, abbreviate the relation $\to_{\mathtt{rsm}}^* \cdot \to_{\mathtt{R/rsm}}^m$ by $\to^m$ for all $m \in \mathbb{N}$. Below, we tacitly employ $\to^{m_1} \cdot \to^{m_2} = \to^{m_1 + m_2}$ for all $m_1, m_2 \in \mathbb{N}$. The proof is by induction on the deduction Π of the statement $([D_1]_H, [e]_{H_1}) \Downarrow_m (C, v)$.

1. Suppose that the last rule in Π has the form:

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

   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 $i \leq k$,

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

   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 $i+1$. Induction hypothesis yields a well-formed configuration $(D_i, H_i, E[e_{i+1}])$ for $E = \mathbf{c}(\ell_1, \dots, \ell_i, \Box, e_{i+2}, \dots, e_k)$ reachable by a Derivation ($\dagger$). As the configuration (D_i, H_i, e_{i+1}) unfolds to (C_i, t_{i+1}), the induction hypothesis of the claim on the assumption $(C_i, t_{i+1}) \Downarrow_{m_{i+1}} (C_{i+1}, v_{i+1})$ yields $(D_i, H_i, e_{i+1}) \to^{m_{i+1}} (D_{i+i}, H_{i+1}, \ell_{i+1})$ where the resulting configuration unfolds to (C_{i+1}, v_{i+1}). Lifting this reduction to the evaluation context E, we get

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

   Note that we can also lift the equality $[\ell_{i+1}]_{H_{i+1}} = v_{i+1}$ to

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

   with the help of Lemma 4.5(2). As we already observed $[D_{i+1}]_{H_{i+1}} = C_{i+1}$, we conclude the Reduction ($\dagger$).

   In total, we thus obtain a reduction $(D_0, H_0, \mathbf{c}(e_1, \dots, e_k)) \to^m (D_k, H_k, \mathbf{c}(\ell_1, \dots, \ell_k))$ 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 $\to^m \cdot \to_{\mathtt{m}}^* = \to^m$ we conclude the case with Lemma 4.6.

2. Suppose that the last rule in Π has the form:

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

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

$$(D_0, H_0, e_0) \to^{\sum_{i=1}^{k} m_i} (D_k, H_k, \mathsf{f}(\ell_1, \ldots, \ell_k)) .$$

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

$$(D_k, H_k, \mathsf{f}(\ell_1, \ldots, \ell_k)) \to^{n} (D_{k+1}, H_{k+1}, \ell) .$$

Summing up we conclude the case.

3. Suppose that the last rule in Π has the form:

$$\frac{(\mathsf{f}(v_1, \ldots, v_k), u) \in C}{(C, \mathsf{f}(v_1, \ldots, v_k)) \Downarrow_0 (C, v)} (\mathtt{read}) .$$

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

$$(D, H, \mathsf{f}(e_1, \ldots, e_k)) \to_{\mathtt{m}}^{*} (D, H_1, \mathsf{f}(\ell_1, e_2, \ldots, e_k)) \to_{\mathtt{m}}^{*} \cdots \to_{\mathtt{m}}^{*} (D, H_k, \mathsf{f}(\ell_1, \ldots, \ell_k)) ,$$

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

$$(D, H, e) = (D, H, \mathsf{f}(e_1, \ldots, e_k)) \to_{\mathtt{m}}^{*} (D, H_k, \mathsf{f}(\ell_1, \ldots, \ell_k)) \to_{\mathtt{r}} (D, H_k, \ell) ,$$

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

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

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

Fix a proper configuration (D, H, e) that unfolds to $(C, \mathsf{f}(v_1, \ldots, v_k))$, in particular $e = \mathsf{f}(e_1, \ldots, e_k)$. As above, we see $(D, H, e) \to_{\mathtt{m}}^{*} (D, H_k, \mathsf{f}(\ell_1, \ldots, \ell_k))$ for a configuration $(D, H_k, \mathsf{f}(\ell_1, \ldots, \ell_k))$ also unfolding to $(C, \mathsf{f}(v_1, \ldots, v_k))$. As $(\mathsf{f}(v_1, \ldots, v_k), v) \notin C$, we have $(\mathsf{f}(\ell_1, \ldots, \ell_k), \ell) \notin D_k$ for any location ℓ, by Lemma 4.4. Since Proposition 3.9 on the assumption yields $\triangle(\mathsf{f}(l_1, \ldots, l_k)) \geqslant_m \mathsf{f}(H\lfloor\ell_1, \ldots, H\lfloor\ell_k)$ for a matching morphism m, in total we obtain

$$(D, H, e) \to_{\mathtt{m}}^{*} (D, H, \mathsf{f}(\ell_1, \ldots, \ell_k)) \to_{\mathtt{R}} (D, H_k, \langle \mathsf{f}(\ell_1, \ldots, \ell_k), r\sigma_m \rangle) .$$

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 $(D, H_k, \langle \mathsf{f}(\ell_1, \ldots, \ell_k), r\sigma_m \rangle)$ unfolds to $(C, r\sigma)$. The induction hypothesis yields a well-formed configuration (D', G, ℓ) unfolding to (C', v) with $(D, H_k, r\sigma_m) \to^m (D', G, \ell)$. Thus

$$\begin{aligned}(D, H_k, \langle \mathsf{f}(\ell_1, \ldots, \ell_k), r\sigma_m \rangle) \to^m & (D', G, \langle \mathsf{f}(\ell_1, \ldots, \ell_k), \ell \rangle) \\ & \to_{\mathtt{s}} (D' \cup \{(\mathsf{f}(\ell_1, \ldots, \ell_k), \ell)\}, G, \ell) .\end{aligned}$$

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

$$[D' \cup \{(\mathsf{f}(\ell_1, \ldots, \ell_k), \ell)\}]_G = C' \cup \{(\mathsf{f}(v_1, \ldots, v_k), v)\} \,.$$

Putting things together, employing $\to_{\mathtt{m}}^* \cdot \to_{\mathtt{R}} \subseteq \to^1$ and $\to^m \cdot \to_{\mathtt{s}} = \to^m$ we conclude $(D, H, e) \to^{m+1} (D' \cup \{(\mathsf{f}(\ell_1, \ldots, \ell_k), \ell)\}, G, \ell)$, where the resulting configuration unfolds to $(C' \cup \{(\mathsf{f}(v_1, \ldots, v_k), v)\}, v)$. We conclude this final case. □

The following lemma completes the proof of correctness of $\to_{\mathtt{Rrsm}}$. Here, a binary relation $\to$ is called *deterministic on a set* A if $b_1 \leftarrow a \to b_2$ implies $b_1 = b_2$ for all $a \in A$.

Lemma 4.8 (Determinism)**.**

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

Proof. For Assertion 1, fix $\to_r \in \{\to_{\mathtt{R}}, \to_{\mathtt{r}}, \to_{\mathtt{s}}, \to_{\mathtt{m}}\}$. Let (D, H, e) be a well-formed configuration. We show that any *peak* $(D_1, H_1, e_1) \;{}_r\!\leftarrow (D, H, e) \to_r (D_2, H_2, e_2)$ is *trivial*, i.e. $(D_1, H_1, e_1) = (D_2, H_2, e_2)$. Observe that independent on $\to_r$, the evaluation context E in the corresponding rule is unique. From this, we conclude the assertion by case analysis on $\to_r$. The non-trivial cases are $\to_r = \to_{\mathtt{R}}$ and $\to_r = \to_{\mathtt{r}}$. In the former case, we conclude using that rules in $\mathcal{R}$ are non-overlapping, tacitly employing Proposition 3.9. The latter case we conclude using that D is well-formed.

Finally, for Assertion 2 consider a peak $(D_1, H_1, e_1) \;{}_{r_1}\!\leftarrow (D, H, e) \to_{r_2} (D_2, H_2, e_2)$ for two relations $\to_{r_1}, \to_{r_2} \in \{\to_{\mathtt{R}}, \to_{\mathtt{r}}, \to_{\mathtt{s}}, \to_{\mathtt{m}}\}$. We show that this peak is trivial by induction on the expression e. By the previous assertion, it suffices to consider only the case $\to_{r_1} \neq \to_{r_2}$.

The base case constitutes of the cases (i) $e = \mathsf{f}(\ell_1, \ldots, \ell_k)$, (ii) $e = \langle \mathsf{f}(\ell_1, \ldots, \ell_k), \ell \rangle$ and (iii) $e = \mathbf{c}(\ell_1, \ldots, \ell_k)$. The only potential peak can occurs in case (i) between relations $\to_{\mathtt{R}}$ and $\to_{\mathtt{r}}$. Here, a non-trivial peak is prohibited by the pre-conditions put on H. For the inductive step, we consider a peak

$$(D_1, H_1, E[e_1']) \;{}_{r_1}\!\leftarrow (D, H, E[e']) \to_{r_2} (D_2, H_2, E[e_2']) \,,$$

where $e = E[e']$ for a context E. As we thus have a peak $(D_1, H_1, e_1') \;{}_{r_1}\!\leftarrow (D, H, e') \to_{r_2} (D_2, H_2, e_2')$, which by induction hypothesis is trivial, we conclude the assertion. □

Theorem 4.9. *Suppose* $(\varnothing, \mathsf{f}(v_1, \ldots, v_k)) \Downarrow_m (C, v)$ *holds for a reducible term* $\mathsf{f}(v_1, \ldots, v_k)$*. Then for each initial configuration* $(\varnothing, H, e)$ *with* $[e]_H = \mathsf{f}(v_1, \ldots, v_k)$*, there exists a unique sequence* $(\varnothing, H, e) \to_{\mathtt{R/rsm}}^m (D, G, \ell)$ *for a location* ℓ *in* G *with* $[\ell]_G = v$*.*

Proof. As $\mathsf{f}(v_1, \ldots, v_k)$ is reducible, it follows that $m \geq 1$. Hence the theorem follows from Lemma 4.7 and Lemma 4.8. □

4.2. Invariance

Theorem 4.9 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 $|\ell| := 1$, $|f(e_1, \ldots, e_k)| := 1 + \sum_{i=1}^k |e_i|$ and $|\langle \mathsf{f}(\ell_1, \ldots, \ell_k), e \rangle| := 1 + |e|$. In correspondence, the *weight* $\mathsf{wt}(e)$ is defined by ignoring locations, i.e. $\mathsf{wt}(\ell) := 0$. Recall that a reduction $(D_1, H_1, e_1) \to_{\mathtt{R/rsm}}^m (D_2, H_2, e_2)$ consists of m applications of $\to_{\mathtt{R}}$, all possibly interleaved by $\to_{\mathtt{rsm}}$-reductions. As a first step, we thus estimate the overall length of the reduction $(D_1, H_1, e_1) \to_{\mathtt{R/rsm}}^m (D_2, H_2, e_2)$ in m and the size of e_1. Set $\Delta := \max\{|r| \mid l \to r \in \mathcal{R}\}$. The following serves as an intermediate lemma.

Lemma 4.10. *The following properties hold:*

1. *If* $(D_1, H_1, e_1) \to_{\mathtt{rsm}} (D_2, H_2, e_2)$ *then* $\mathsf{wt}(e_2) < \mathsf{wt}(e_1)$.
2. *If* $(D_1, H_1, e_1) \to_{\mathtt{R}} (D_2, H_2, e_2)$ *then* $\mathsf{wt}(e_2) \leq \mathsf{wt}(e_1) + \Delta$.

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

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

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

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

Lemma 4.11. *If* $(D_1, H_1, e_1) \to^m_{\mathtt{R}/\mathtt{rsm}} (D_2, H_2, e_2)$ *then* $(D_1, H_1, e_1) \to^n_{\mathtt{Rrsm}} (D_2, H_2, e_2)$ *for* $n \leq (1 + \Delta) \cdot m + \mathsf{wt}(e)$ *and* $\Delta \in \mathbb{N}$ *a constant depending only on* $\mathtt{P}$.

Proof. For a configuration $c = (D, H, e)$ define $\mathsf{wt}(c) := \mathsf{wt}(e)$ and let Δ be defined as in Lemma 4.10. Consider $(D_1, H_1, e_1) \to^m_{\mathtt{R}/\mathtt{rsm}} (D_2, H_2, e_2)$, which can be written as a reduction

$$(D_1, H_1, e_1) = c_0 \to^{n_0}_{\mathtt{rsm}} d_0 \to_{\mathtt{R}} c_1 \to^{n_1}_{\mathtt{rsm}} d_1 \to_{\mathtt{R}} \cdots \to^{n_m}_{\mathtt{rsm}} d_m \,, \tag{‡}$$

of length $n := m + \sum_{i=0}^{k} n_i$. Lemma 4.10 yields (i) $n_i \leq \mathsf{wt}(c_i) - \mathsf{wt}(d_i)$ for all $0 \leq i \leq m$; and (ii) $\mathsf{wt}(c_{i+1}) - \mathsf{wt}(d_i) \leq \Delta$ for all $0 \leq i < m$. Hence overall, the Reduction (‡) is of length

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

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

Lemma 4.12. *If* $(D_1, H_1, e_1) \to_{\mathtt{Rrsm}} (D_2, H_2, e_2)$ *then* $|(D_2, H_2, e_2)| \leq |(D_1, H_1, e_1)| + \Delta$.

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

Theorem 4.13. *There exists a polynomial* $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *such that for every initial configuration* $(\varnothing, H_1, e_1)$*, a configuration* (D_2, H_2, e_2) *with* $(\varnothing, H_1, e_1) \to^m_{\mathtt{R}/\mathtt{rsm}} (D_2, H_2, e_2)$ *is computable from* $(\varnothing, H_1, e_1)$ *in time* $p(|H_1| + |e_1|, m)$.

Proof. It is tedious, but not difficult to show that the function which implements a step $c \to_{\mathtt{Rrsm}} d$, i.e. which maps $\lceil c \rceil$ to $\lceil d \rceil$, is computable in polynomial time in $\lceil c \rceil$, and thus in the size $|c|$ of the configuration c. Iterating this function at most $n := (1 + \Delta) \cdot m + |(\varnothing, H_1, e_1)|$ times on input $\lceil (\varnothing, H_1, e_1) \rceil$, yields the desired result $\lceil (D_2, H_2, e_2) \rceil$ by Lemma 4.11. Since each iteration increases the size of a configuration by at most the constant Δ (Lemma 4.12), in particular the size of each intermediate configuration is bounded by a linear function in $|(\varnothing, H_1, e_1)| = |H_1| + |e_1|$ and n, the theorem follows. □

Combining Theorem 4.9 and Theorem 4.13 we thus obtain the desired invariance result.

Theorem 4.14 (Invariance of Memoized Runtime Complexity)**.** *There exists a polynomial $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for $(\varnothing, \mathbf{f}(v_1, \ldots, v_k)) \Downarrow_m (C, v)$, the value v represented as graph is computable from $v_1, \ldots, v_k$ in time $p(\sum_{i=1}^{k} |v_i|, m)$.*

Theorem 4.14 thus confirms that the cost m of a reduction $(\varnothing, \mathbf{f}(v_1, \ldots, v_k)) \Downarrow_m (C, v)$ is a suitable cost measure. In other words, the *memoized runtime complexity* of a function $\mathbf{f}$, relating input size $n \in \mathbb{N}$ to the maximal cost m of evaluating $\mathbf{f}$ on arguments $v_1, \ldots, v_k$ of size up to n, i.e. $(\varnothing, \mathbf{f}(v_1, \ldots, v_k)) \Downarrow_m (C, v)$ with $\sum_{i=1}^{k} |v_i| \leq n$, is an *invariant cost model*.

Example 4.15 (Continued from Example 3.6)**.** Reconsider the program $\mathtt{P}_{\mathtt{rabbits}}$ and the evaluation of a call $\boldsymbol{r}(\mathbf{S}^n(\mathbf{0}))$ 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 $\boldsymbol{r}(\mathbf{S}^n(\mathbf{0})) \Downarrow_m v_n$ with $m \leq 2 \cdot n + 1$. Crucially here, the two intermediate functions $\boldsymbol{a}$ and $\boldsymbol{b}$ defined by simultaneous recursion are called only on proper subterms of the input $\mathbf{S}^n(\mathbf{0})$, hence in particular the rules defining $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively, are unfolded at most n times. As a consequence of the bound on m and Theorem 4.14 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 compact in comparison to a linear representation without sharing, it is not exponentially more *succinct* than a reasonable encoding for graphs (e.g. representations as circuits). In such succinct encodings not even equality can be decided in polynomial time. Our form of representation does clearly not fall into this category. In particular, in our setting it can be easily checked in polynomial time that two DAGs represent the same value.

5. GRSR is Sound for Polynomial Time

Sometimes (e.g., in [20]), 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* $\|v_1, \ldots, v_k\|$ for a *sequence* of elements $v_1, \ldots, v_k \in \mathbb{A}$ is defined as the number of subterms in $v_1, \ldots, v_k$, i.e. the cardinality of the set $\bigcup_{1 \leq i \leq k} \mathsf{STs}(v_i)$. Then $\|v_1, \ldots, v_k\|$ corresponds to the number of locations necessary to store the values $v_1, \ldots, v_k$ on a heap (compare Lemma 4.4). If $\mathbf{A}$ is the expression $\mathbb{A}_{n_1} \times \ldots \times \mathbb{A}_{n_m}$, n is a natural number, and $\vec{t}$ is a sequence of m terms, then $\vec{t}_{>n}$ is defined to be $t_{i_1}, \ldots, t_{i_k}$, where $i_1, \ldots, i_k$ are precisely those indices such that $n_{i_1}, \ldots, n_{i_k} > n$. Similarly for $\vec{t}_{=n}$, and for expressions like $\|\vec{t}\|_{\mathbf{A}}^{\geq n}$ and $\|\vec{t}\|_{\mathbf{A}}^{=n}$.

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

Proof. The following strengthening of the statement above can be proved by induction on the structure of the proof of $\vec{\mathtt{f}} \triangleright \mathbf{A} \to \mathbb{A}_n$: for every *sequence* of functions $\vec{\mathtt{f}}$ such that $\vec{\mathtt{f}} \triangleright \mathbf{A} \to \mathbb{A}_n$, there is a monotone polynomial $p_{\vec{\mathtt{f}}}$ such that for all $\vec{t}$ it holds that

$$\mathsf{STs}(\vec{\mathtt{f}}(\vec{t})) \subseteq \mathsf{STs}(\vec{t}_{=n}) \cup NC^{\vec{\mathtt{f}}}_{\vec{t}_{>n}} ,$$

for some set $NC^{\vec{\mathtt{f}}}_{\vec{t}_{>n}}$ with $|NC^{\vec{\mathtt{f}}}_{\vec{t}_{>n}}| \leq p_{\vec{\mathtt{f}}}(\|t\|_{\mathbf{A}}^{\geq n})$. Let us consider some interesting cases:

- If one element $\mathtt{f}_i$ in the sequence $\vec{\mathtt{f}}$, is either $\mathtt{f_c}$ or Π^m_k, then one could handle it separately, apply the induction hypothesis to the rest of the sequence, and assemble the results.
- If one element $\mathtt{f}_i$ in the sequence $\vec{\mathtt{f}}$, is in the form $\mathtt{g} \circ (\mathtt{h}_1, \ldots, \mathtt{h}_k)$, then, again one could handle it separately, apply the induction hypothesis to the rest of the sequence, and assemble the results.
- The interesting case is the one in which the functions in $\vec{\mathtt{f}}$ are obtained through a single instance of simultaneous primitive recursion. For the sake of readability, let us consider the case $\vec{\mathtt{f}} = \mathtt{f}_1, \mathtt{f}_2$, where $\mathbf{A} = \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_0$, $n = 0$ and the algebra $\mathbb{A}$ includes just two constructors, namely **nil**, having arity 0, and **bin**, having arity 2. There are thus four functions $\mathtt{g}_1, \mathtt{g}_2, \mathtt{h}_1, \mathtt{h}_2$ such that

$$\vec{\mathtt{g}} \triangleright \mathbb{A}_1 \times \mathbb{A}_0 \to \mathbb{A}_0 \ ;$$
$$\vec{\mathtt{h}} \triangleright \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_0 \times \mathbb{A}_0 \times \mathbb{A}_0 \times \mathbb{A}_0 \to \mathbb{A}_0 \ ;$$

and, moreover:

$$\mathtt{f}_i(\mathbf{nil}, x, y) = \mathtt{g}_i(x, y) \ ;$$
$$\mathtt{f}_i(\mathbf{bin}(z, w), x, y) = \mathtt{h}_i(z, w, x, \mathtt{f}_1(z, x, y), \mathtt{f}_1(w, x, y), \mathtt{f}_2(z, x, y), \mathtt{f}_2(w, x, y), y) \ .$$

From the IH, we know that there exist appropriate polynomials $p_{\vec{\mathtt{g}}}$ and $p_{\vec{\mathtt{h}}}$ and sets $NC^{\vec{\mathtt{g}}}_t$ and $NC^{\vec{\mathtt{g}}}_{t,s,q}$ for all terms t, s, r. Let us then define

$$NC^{\vec{\mathtt{f}}}_{t,s} = NC^{\vec{\mathtt{g}}}_s \cup \bigcup_{\mathbf{bin}(r,q) \in \mathsf{STs}(t)} NC^{\vec{\mathtt{h}}}_{r,q,s} \ ;$$
$$p_{\vec{\mathtt{f}}}(x) = p_{\vec{\mathtt{g}}}(x) + x \cdot p_{\vec{\mathtt{h}}}(x).$$

First of all, one easily realizes that

$$\begin{aligned}
|NC^{\vec{\mathtt{f}}}_{t,s}| &\leq |NC^{\vec{\mathtt{g}}}_s| + |\bigcup_{\mathbf{bin}(r,q) \in \mathsf{STs}(t)} NC^{\vec{\mathtt{h}}}_{r,q,s}| \\
&\leq p_{\vec{\mathtt{g}}}(\|s\|) + \sum_{\mathbf{bin}(r,q) \in \mathsf{STs}(t)} p_{\vec{\mathtt{h}}}(\|r, q, s\|) \leq p_{\vec{\mathtt{g}}}(\|s\|) + \sum_{\mathbf{bin}(r,q) \in \mathsf{STs}(t)} p_{\vec{\mathtt{h}}}(\|\mathbf{bin}(r, q), s\|) \\
&\leq p_{\vec{\mathtt{g}}}(\|s\|) + \sum_{\mathbf{bin}(r,q) \in \mathsf{STs}(t)} p_{\vec{\mathtt{h}}}(\|t, s\|) \leq p_{\vec{\mathtt{g}}}(\|s\|) + \|t\| \cdot p_{\vec{\mathtt{h}}}(\|t, s\|) \ .
\end{aligned}$$

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

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

This concludes the proof. □

Once we know that ramified recursive definitions are not too fast-growing for the minimal shared size, we know that all terms around do not have a too-big minimal shared size. There is still some work to do to get to our goal. It is convenient to introduce another auxiliary concept before proceeding. If $\mathtt{f}$ is a function on n arguments in SimRec($\mathbb{A}$) and $v_1, \ldots, v_n$ are elements of $\mathbb{A}$, we define $FC^{\mathtt{f}}_{v_1,\ldots,v_n}$ as follows, by induction:

$$
\begin{aligned}
FC^{\mathtt{f_c}}_{v,\ldots,v_n} &= \{(\mathtt{f_c}, v_1, \ldots, v_n)\} \; ; \\
FC^{\Pi^m_n}_{v_1,\ldots,v_n} &= \{(\Pi^m_n, v_1, \ldots, v_n)\} \; ; \\
FC^{\mathtt{f}\circ(\mathtt{g}_1,\ldots,\mathtt{g}_m)}_{v_1,\ldots,v_n} &= \bigcup_{i=1}^{m} FC^{\mathtt{g}_i}_{v_1,\ldots,v_n} \cup FC^{\mathtt{f}}_{\mathtt{g}_1(\vec{v}),\ldots,\mathtt{g}_n(\vec{v})} \cup \{(\mathtt{f}\circ(\mathtt{g}_1,\ldots,\mathtt{g}_m), v_1, \ldots, v_n)\} \; ; \\
FC^{case(\{\mathtt{f}_i\}_{1\leq i\leq k})}_{\mathbf{c}_i(u_1,\ldots,u_{\mathsf{ar}(\mathbf{c}_i)}),v_2,\ldots,v_n} &= FC^{\mathtt{f}_i}_{u_1,\ldots,u_{\mathsf{ar}(\mathbf{c}_i)},v_2,\ldots,v_n} \cup \{(case(\{\mathtt{f}_i\}_{1\leq i\leq k}), \mathbf{c}_i(u_1, \ldots, u_{\mathsf{ar}(\mathbf{c}_i)}), v_2, \ldots, v_n)\} \; ; \\
FC^{\mathtt{g}_j}_{\mathbf{c}_i(u_1,\ldots,u_{\mathsf{ar}(\mathbf{c}_i)}),v_2,\ldots,v_n} &= \bigcup_{i=1}^{\mathsf{ar}(\mathbf{c}_i)} \bigcup_{j=1}^{m} FC^{\mathtt{g}_j}_{u_i,v_2,\ldots,v_n} \cup FC^{\mathtt{f}_{i,j}}_{u_1,\ldots,u_{\mathsf{ar}(\mathbf{c}_i)},\mathtt{g}_1(u_1,v_2,\ldots,v_n),\ldots,\mathtt{g}_m(u_{\mathsf{ar}(\mathbf{c}_i)},v_2,\ldots,v_n),v_2,\ldots,v_n} \\
&\qquad \cup \{(\mathtt{g}_j, \mathbf{c}_i(u_1, \ldots, u_{\mathsf{ar}(\mathbf{c}_i)}), v_2, \ldots, v_n)\} \\
&\qquad\qquad \text{where } \{\mathtt{g}_j\}_{1\leq j\leq n} = simrec(\{\mathtt{f}_{i,j}\}_{1\leq i\leq k, 1\leq j\leq m}).
\end{aligned}
$$

Please observe how subsets of $FC^{\mathtt{f}}_{\vec{v}}$ are in natural correspondence to (proper) caches for the program $\mathtt{P_f}$, which justifies that we identify sets $FC^{\mathtt{f}}_{\vec{v}}$ with caches $\{(\boldsymbol{g}(\vec{v}), u) \mid (\mathtt{g}, \vec{v}) \in FC^{\mathtt{f}}_{\vec{v}} \text{ and } \boldsymbol{f}(\vec{v}) \downarrow u\}$. The parameter we have just introduced is all that is needed to estimate the memoized runtime complexity.

Lemma 5.2. *If $\mathtt{f}$ is a function on k arguments in* SimRec($\mathbb{A}$)*, then for every $v_1, \ldots, v_k \in \mathbb{A}$ and for every proper cache C, it holds that $(C, \boldsymbol{f}(v_1, \ldots, v_k)) \Downarrow_m (D, u)$ with respect to the program $\mathtt{P_f}$ associated with $\mathtt{f}$, where $m \leq |FC^{\mathtt{f}}_{\vec{v}}|$.*

Proof. We prove the following strengthening of the thesis, by induction on the proof that $\mathtt{f}$ is in SimRec($\mathbb{A}$): $m \leqslant |FC^{\mathtt{f}}_{\vec{v}} \setminus C|$ and $D = C \cup FC^{\mathtt{f}}_{\vec{v}}$. We consider the most involved case, namely where the considered function is defined by simultaneous recursion. To avoid notational clutter, we again consider a special case of two functions $\mathtt{f}_1, \mathtt{f}_2$ over $\mathbb{A} = \{\mathbf{nil}, \mathbf{bin}\}$ defined by simultaneous recursion via the equations

$$
\begin{aligned}
\mathtt{f}_i(\mathbf{nil}, x) &= \mathtt{g}_i(x) \; ; \\
\mathtt{f}_i(\mathbf{bin}(z, w), x) &= \mathtt{h}_i(z, w, x, \mathtt{f}_1(z, x), \mathtt{f}_1(w, x), \mathtt{f}_2(z, x), \mathtt{f}_2(w, x)) \; .
\end{aligned}
$$

By induction hypothesis the property holds for the functions $\mathtt{g}_1, \mathtt{g}_2 : \mathbb{A} \to \mathbb{A}$ and $\mathtt{h}_1, \mathtt{h}_2 : \mathbb{A}^5 \to \mathbb{A}$. To conclude the case, we fix $i \in \{1, 2\}$ and proceed by structural induction on the recursion parameter of $\mathtt{f}_i$. Let C be a proper cache.

- In the base case, we consider the statement $(C, \boldsymbol{f}_i(\mathbf{nil}, v)) \Downarrow_m (D, u)$. There are two cases. If $\boldsymbol{f}_i(\mathbf{nil}, v)$ is cached, i.e. $(\boldsymbol{f}_i(\mathbf{nil}, v), u) \in C$, then $m = 0$ and the case follows trivially. For the case that $\boldsymbol{f}_i(\mathbf{nil}, v)$ is not cached, the program $\mathtt{P_f}$ contains the rule $\boldsymbol{f}_i(\mathbf{nil}, x) \to \boldsymbol{g}_i(x)$ by construction. Then by one application of the hypothesis of the outer induction we see

$$
\dfrac{\dfrac{\vdots}{(C, \boldsymbol{g}_i(v)) \Downarrow_m (C \cup FC^{\mathtt{g}_i}_{v}, u)}\,(\mathtt{IH})}{(C, \boldsymbol{f}_i(\mathbf{nil}, v)) \Downarrow_{m+1} (C \cup FC^{\mathtt{g}_i}_{v} \cup \{(\boldsymbol{f}_i(\mathbf{nil}, v), u)\}, u)}\,(\mathtt{update})
$$

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

- For the inductive step, we consider the statement $(C, \boldsymbol{f}_i(\mathbf{bin}(v_1, v_2), v)) \Downarrow_m (D, u)$. It suffices to consider the case where the call $\boldsymbol{f}_i(\mathbf{bin}(v_1, v_2), v)$ is not cached. Abbreviate the sequence of values $v_1, v_2, v, u_1, u_2, u_3, u_4$ by $\vec{u}$. Furthermore, we introduce following abbreviations for caches that arise during the computation of $\boldsymbol{f}_i(\mathbf{bin}(v_1, v_2), v)$:

$$C_1 := C \cup FC^{\mathtt{f}_1}_{v_1,v} \quad C_2 := C_1 \cup FC^{\mathtt{f}_1}_{v_2,v} \quad C_3 := C_2 \cup FC^{\mathtt{f}_2}_{v_1,v} \quad C_4 := C_3 \cup FC^{\mathtt{f}_2}_{v_2,v} \quad C_5 := C_4 \cup FC^{\mathtt{g}_i}_{\vec{u}} \ .$$

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

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

where $D = C_5 \cup \{(\boldsymbol{f}_i(\mathbf{bin}(v_1, v_2), v), u)\}$. Let $C_0 = C$. Note that the sets $C_i \setminus C_{i-1}$ $(i = 1, \ldots, 5)$ and C are all pairwise disjoint, and thus we can write $C_5 \setminus C = \biguplus_{i=1}^5 (C_i \setminus C_{i-1})$. Since induction hypothesis gives $n_i \leqslant |C_i \setminus C_{i-1}|$, we conclude

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

As by definition we have

$$\begin{aligned} FC^{\mathtt{f}_i}_{\mathbf{bin}(v_1,v_2),v} &= FC^{\mathtt{f}_1}_{v_1,v} \cup FC^{\mathtt{f}_1}_{v_2,v} \cup FC^{\mathtt{f}_2}_{v_1,v} \cup FC^{\mathtt{f}_2}_{v_2,v} \cup FC^{\mathtt{g}_i}_{\vec{u}} \cup \{(\boldsymbol{f}_i, \mathbf{bin}(v_1, v_2), v\} \ ; \\ C_5 &= FC^{\mathtt{f}_1}_{v_1,v} \cup FC^{\mathtt{f}_1}_{v_2,v} \cup FC^{\mathtt{f}_2}_{v_1,v} \cup FC^{\mathtt{f}_2}_{v_2,v} \cup FC^{\mathtt{g}_i}_{\vec{u}} \cup C \ , \end{aligned}$$

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

$$1 + \textstyle\sum_{i=1}^5 n_i \leqslant 1 + |C_5 \setminus C| = |FC^{\mathtt{f}_i}_{\mathbf{bin}(v_1,v_2),v} \setminus C| \ .$$

□

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

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

Proof. This is an induction on the structure of the proof of $\mathtt{f} \triangleright \mathbf{A} \to \mathbb{A}_n$. Let us consider the most interesting cases:

- Suppose $\mathtt{f}$ is defined by simultaneous recursion on notation. For the sake of readability, let us consider the case $\mathtt{f} = \mathtt{f}_1$ and $\mathtt{f}_2$ are defined by simultaneous recursion, $\mathbf{A} = \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_0$, $n = 0$ and the algebra $\mathbb{A}$ includes just two constructors, namely **nil**, having arity 0, and **bin**, having arity 2. There are thus four functions $\vec{\mathtt{g}} := \mathtt{g}_1, \mathtt{g}_2$, $\vec{\mathtt{h}} := \mathtt{h}_1, \mathtt{h}_2$ such that

$$\begin{aligned} &\vec{\mathtt{g}} \triangleright \mathbb{A}_1 \times \mathbb{A}_0 \to \mathbb{A}_0 \ ; \\ &\vec{\mathtt{h}} \triangleright \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_1 \times \mathbb{A}_0 \times \mathbb{A}_0 \times \mathbb{A}_0 \times \mathbb{A}_0 \times \mathbb{A}_0 \to \mathbb{A}_0 \ , \end{aligned}$$

and, moreover:

$$\begin{aligned} \mathtt{f}_i(\mathbf{nil}, x, y) &= \mathtt{g}_i(x, y) \ ; \\ \mathtt{f}_i(\mathbf{bin}(z, w), x, y) &= \mathtt{h}_i(z, w, x, \mathtt{f}_1(z, x, y), \mathtt{f}_1(w, x, y), \mathtt{f}_2(z, x, y), \mathtt{f}_2(w, x, y), y) \ . \end{aligned}$$

From the IH, we know that there exist appropriate polynomials $p_{\mathtt{g}_i}$ and $p_{\mathtt{h}_i}$. Please observe that

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

where expressions like $FC^{\vec{\mathtt{e}}}_{\vec{t}}$ stand for the union $FC^{\mathtt{e}_1}_{\vec{t}} \cup FC^{\mathtt{e}_2}_{\vec{t}}$. This can be easily proved by induction on v, and is an easy consequence of the way $FC^{\cdot}_{\cdot}$ is defined. Now, define

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

where expressions like $p_{\vec{\mathtt{e}}}$ stand for $p_{\mathtt{e}_1} + p_{\mathtt{e}_2}$ and $q_{\vec{\mathtt{f}}}$ is the polynomial from Lemma 5.1. Let us prove that this is, indeed, a correct bound:

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

This concludes the proof. □

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

The following, then, is just a corollary of Lemma 5.2, Lemma 5.4 and Invariance (Theorem 4.14).

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

Example 5.6 (Continued from Example 4.15)**.** In Example 3.4 we already indicated that the function $\mathtt{rabbits} : \mathbb{N} \to \mathbb{T}$ 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 [21–23] and term rewrite systems [24, 25] could be proved. Several works (e.g. [16, 17, 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, Perspectives in Computer Science, 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 1$^{\text{st}}$ International Workshop on Developments in Implicit Complexity, vol. 23 of *Electronic Proceedings in Theoretical Computer Science*, 47–62, 2010.

[3] D. Leivant, Stratified Functional Programs and Computational Complexity, in: Proc. of 20$^{\text{th}}$ Annual Symposium on Principles of Programming Languages, Association for Computing Machinery, 325–333, 1993.

[4] S. Bellantoni, Predicative Recursion and Computational Complexity, Ph.D. thesis, University of Toronto, 1992.

[5] S. Bellantoni, Predicative Recursion and the Polytime Hierarchy, in: Feasible Mathematics II, Perspectives in Computer Science, Birkhäuser Science, 1994.

[6] I. Oitavem, Implicit Characterizations of Pspace., in: Proof Theory in Computer Science, 170–190, 2001.

[7] T. Arai, N. Eguchi, A New Function Algebra of EXPTIME Functions by Safe Nested Recursion, ACM Transactions on Computational Logic 10 (4).

[8] G. Bonfante, R. Kahle, J.-Y. Marion, I. Oitavem, Recursion Schemata for NCk, in: Proc. of 22$^{\text{nd}}$ European Association for Computer Science Logic, vol. 5213 of *Lecture Notes in Computer Science*, Springer Verlag Heidelberg, 49–63, 2008.

[9] N. Danner, J. S. Royer, Ramified Structural Recursion and Corecursion, CoRR abs/1201.4567.

[10] B. Hoffmann, Term Rewriting with Sharing and Memoization, in: Proc. of 3$^{\text{rd}}$ Algebraic and Logic Programming, vol. 632 of *Lecture Notes in Computer Science*, Springer Verlag Heidelberg, 128–142, 1992.

[11] M. Avanzini, U. Dal Lago, On Sharing, Memoization, and Polynomial Time, in: Proc. of 32$^{\text{nd}}$ International Symposium on Theoretical Aspects of Computer Science, vol. 30 of *Leibniz International Proceedings in Informatics*, Leibniz-Zentrum fr Informatik, 62–75, 2015.

[12] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[13] E. A. Cichon, A. Weiermann, Term Rewriting Theory for the Primitive Recursive Functions, Annals of Pure and Applied Logic 83 (3) (1997) 199–223.

[14] D. Plump, Essentials of Term Graph Rewriting, Electronic Notes in Theoretical Computer Science 51 (2001) 277–289.

[15] 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.

[16] J.-Y. Marion, Analysing the Implicit Complexity of Programs, Information and Computation 183 (2003) 2–18.

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

[18] N. Hirokawa, G. Moser, Automated Complexity Analysis Based on the Dependency Pair Method, in: Proc. of 4$^{\text{th}}$ International Joint Conference on Automated Reasoning, vol. 5195 of *Lecture Notes in Artificial Intelligence*, Springer Verlag Heidelberg, 364–380, 2008.

[20] S. Bellantoni, S. Cook, A new Recursion-Theoretic Characterization of the Polytime Functions, Computational Complexity 2 (2) (1992) 97–110.

[21] B. Accattoli, U. Dal Lago, On the Invariance of the Unitary Cost Model for Head Reduction, in: Proc. of 23$^{\text{rd}}$ International Conference on Rewriting Techniques and Applications, vol. 15 of *Leibniz International Proceedings in Informatics*, Leibniz-Zentrum fr Informatik, 22–37, 2012.

[22] U. Dal Lago, S. Martini, On Constructor Rewrite Systems and the Lambda Calculus, Logical Methods in Computer Science 8 (3) (2012) 1–27.

[23] B. Accattoli, U. Dal Lago, Beta Reduction is Invariant, Indeed, in: Proc. of 23$^{\text{rd}}$ European Association for Computer Science Logic, Association for Computing Machinery, 8:1–8:10, 2014.

[24] 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.

[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 fr Informatik, 33–48, 2010.

[26] P. Baillot, U. Dal Lago, J. Moyen, On Quasi-interpretations, Blind Abstractions and Implicit Complexity, Mathematical Structures in Computer Science 22 (4) (2012) 549–580.