The Complexity of the Diagonal Problem for Recursion Schemes*

Paweł Parys

University of Warsaw, Poland

Abstract

We consider nondeterministic higher-order recursion schemes as recognizers of languages of finite words or finite trees. We establish the complexity of the diagonal problem for schemes: given a set of letters A and a scheme $\mathcal{G}$, is it the case that for every number n the scheme accepts a word (a tree) in which every letter from A appears at least n times. We prove that this problem is $(m-1)$-EXPTIME-complete for word-recognizing schemes of order m, and m-EXPTIME-complete for tree-recognizing schemes of order m.

1998 ACM Subject Classification F.4.3 Formal Languages

Keywords and phrases diagonal problem, higher-order recursion schemes, intersection types, downward closure

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.45

1 Introduction

The *diagonal problem* in its original formulation over finite words asks, for a set of letters A and a language of words L, whether for every $n \in \mathbb{N}$ there is a word in L where every letter from A occurs at least n times. The same problem can be also considered for a language of finite trees. In this paper, we study the complexity of the diagonal problem for languages of finite words and finite trees recognized by nondeterministic higher-order recursion schemes.

Higher-order recursion schemes (*schemes* in short) are used to faithfully represent the control flow of programs in languages with higher-order functions. This formalism is equivalent via direct translations to simply-typed λY-calculus [22] and to higher-order OI grammars [8, 16]. Collapsible pushdown systems [9] and ordered tree-pushdown systems [5] are other equivalent formalisms. Schemes cover some other models such as indexed grammars [1] and ordered multi-pushdown automata [3].

The goal of this paper is to establish the complexity of the diagonal problem for higher-order schemes. By a recent result by Clemente et al. [6] we know that this problem is decidable. For schemes of order m their algorithm works in $f(m)$-fold exponential time for some quadratic function f (although the complexity of the algorithm is not mentioned explicitly in the paper, it can be easily estimated as being such). In the current work, we tighten the upper bound: we prove that the diagonal problem for word-recognizing (tree-recognizing) schemes of order m is $(m-1)$-EXPTIME-complete (m-EXPTIME-complete, respectively).

Let us recall from [6] that the decidability result for the diagonal problem entailed other decidability results for recursion schemes, concerning in particular computability of the downward closure of recognized languages [23], and the problem of separability by piecewise testable languages [7]. Although our complexity result for the diagonal problem does not

* Work supported by the National Science Centre, Poland (grant no. 2016/22/E/ST6/00041).

influence directly our knowledge on the complexity of the other problems (the aforementioned reductions preserve only decidability, but not complexity), it can be seen as the first step in establishing the complexity of the other problems as well.

Our solution is based on an appropriate system of intersection types. Intersection types were intensively used in the context of schemes, for several purposes like model-checking [11, 14, 4, 21], pumping [12], transformations of schemes [13, 6], etc. Among such type systems we have to distinguish those [18, 19], in which the (appropriately defined) size of a type derivation for a term approximates some quantity visible in the Böhm tree of that term. In particular, in our recent work [19] we have developed a type system that allows to solve the diagonal problem for a special case of a single-letter alphabet.

Here, we generalize the last type system mentioned above to multiple letters. In result, a type derivation in this system is labeled by flags of different kinds. The key property lies in some (quite rough) correspondence between words (trees) that can be generated from a term and type derivations for the term, where, for every letter a, the number of appearances of a in the generated word (tree) is approximated by the number of appearances of an appropriate flag in the type derivation. In effect, the diagonal problem reduces to checking whether there exist type derivations with arbitrarily many flags corresponding to every letter from the input set A.

Some further work was needed to carefully optimize the developed type system in order to obtain an algorithm achieving the optimal complexity.

Our paper is structured as follows. In Section 2 we introduce all necessary definitions. In Section 3 we introduce the type system, and we show how to use it for deciding the diagonal problem for word-recognizing schemes. Finally, Section 4 presents extensions of the algorithm, in particular to tree-recognizing schemes.

2 Preliminaries

Infinitary λ-calculus. The set of *sorts* (a/k/a simple types), constructed from a unique basic sort o using a binary operation $\to$, is defined as usual. We omit brackets on the right of an arrow, so e.g. $o\to(o\to o)$ is abbreviated to $o\to o\to o$. The order of a sort is defined by induction: $ord(o) = 0$, and $ord(\alpha_1\to\dots\to\alpha_s\to o) = 1 + \max(ord(\alpha_1), \dots, ord(\alpha_s))$ for $s \geq 1$.

A sort $\alpha_1\to\dots\to\alpha_s\to o$ is *homogeneous* if $ord(\alpha_1) \geq \dots \geq ord(\alpha_s)$ and all $\alpha_1, \dots, \alpha_s$ are homogeneous. In the sequel we restrict ourselves to homogeneous sorts (even if not always this is written explicitly).

Let Σ be an infinite set of symbols (alphabet). To denote nondeterministic choices we use a symbol nd. Assuming that $\mathsf{nd} \notin \Sigma$, we denote $\Sigma^{\mathsf{nd}} = \Sigma\cup\{\mathsf{nd}\}$. Let also $\mathcal{V} = \{x^\alpha, y^\beta, z^\gamma, \dots\}$ be a set of variables, containing infinitely many variables of every homogeneous sort (sort of a variable is written in superscript).

We consider infinitary, sorted λ-calculus. *Infinitary λ-terms* (or just *λ-terms*) are defined by coinduction, according to the following rules:

- node constructor – if $a \in \Sigma^{\mathsf{nd}}$, and $P_1^o, \dots, P_r^o$ are λ-terms, then $(a\langle P_1^o, \dots, P_r^o\rangle)^o$ is a λ-term,[1]

[1] Our node constructor differs from the standard definition in two aspects. First, one usually assumes that symbols are ranked, i.e., that the number r is determined by the choice of a. Second, typically a symbol a is considered itself as a λ-term of sort $\underbrace{o\to\dots\to o}_{r}\to o$, which after applying $P_1^o, \dots, P_r^o$ as arguments is equivalent to our $(a\langle P_1^o, \dots, P_r^o\rangle)^o$. These are, though, only superficial differences.

- variable – every variable $x^\alpha \in \mathcal{V}$ is a λ-term,
- application – if $P^{\alpha\to\beta}$ and Q^α are λ-terms, then $(P^{\alpha\to\beta}\,Q^\alpha)^\beta$ is a λ-term, and
- λ-binder – if P^β is a λ-term and x^α is a variable, then $(\lambda x^\alpha.P^\beta)^{\alpha\to\beta}$ is a λ-term;

in the above, α, β, and $\alpha\to\beta$ are homogeneous sorts. We naturally identify λ-terms differing only in names of bound variables. We often omit the sort annotations of λ-terms, but we keep in mind that every λ-term (and every variable) has a particular sort. *Free variables* of a λ-term are defined as usual. A λ-term P is *closed* if it has no free variables.

For a λ-term P, the *order* of P is just the order of its sort, while the *complexity* of P is the smallest number m such that the order of all subterms of P is at most m. We restrict ourselves to λ-terms that have finite complexity. We also define the order of a β-reduction as the order of the involved variable. More precisely, for a number $k \in \mathbb{N}$, we say that there is a β-reduction of order k from a λ-term P to a λ-term Q, written $P \to_{\beta(k)} Q$, if Q is obtainable from P by replacing a redex $(\lambda x.R)\,S$ where $ord(x) = k$ with $R[S/x]$.

Trees. A *tree* is defined as a λ-term that is built using only node constructors, i.e., not using variables, applications, nor λ-binders. A tree is Γ-*labeled* if only symbols from Γ appear in it.

Let us now define how we resolve nondeterministic choices. Although this is mainly used for trees, we define it for arbitrary λ-terms. We write $P \to_{\mathsf{nd}} Q$ if Q is obtained from P by choosing some appearance of the nd symbol surrounded only by symbols from Σ, and removing this nd symbol together with all but one of its arguments. Formally, we let $\to_{\mathsf{nd}}$ to be the smallest relation such that $\mathsf{nd}\langle P_1,\dots,P_r\rangle \to_{\mathsf{nd}} P_i$ for $i \in \{1,\dots,r\}$, and if $a \in \Sigma$, and $P_i \to_{\mathsf{nd}} P'_i$ for some $i \in \{1,\dots,r\}$, and $P_j = P'_j$ for all $j \in \{1,\dots,r\} \setminus \{i\}$, then $a\langle P_1,\dots,P_r\rangle \to_{\mathsf{nd}} a\langle P'_1,\dots,P'_r\rangle$. For a relation r, by r^* we denote the reflexive transitive closure of r. For a λ-term P (which is usually a Σ^{nd}-labeled, potentially infinite tree), by $\mathcal{L}(P)$ we denote the set of all finite, Σ-labeled trees T such that $P \to_{\mathsf{nd}}^* T$.

Böhm Trees. We consider Böhm trees only for closed λ-terms of sort o. For such a term P, its *Böhm tree* $BT(P)$ is constructed by coinduction, as follows: if there is a sequence of β-reductions from P to a λ-term of the form $a\langle P_1,\dots,P_r\rangle$ (where $a \in \Sigma^{\mathsf{nd}}$), then $BT(P) = a\langle BT(P_1),\dots,BT(P_r)\rangle$; otherwise $BT(P) = \mathsf{nd}\langle\rangle$.

Higher-Order Recursion Schemes. We use a very loose definition of schemes. A *higher-order recursion scheme* (or just a *scheme*) is a triple $\mathcal{G} = (\mathcal{N}, \mathcal{R}, N_0^o)$, where $\mathcal{N} \subseteq \mathcal{V}$ is a finite set of nonterminals, $\mathcal{R}$ is a function that maps every nonterminal $N \in \mathcal{N}$ to a finite λ-term whose free variables are contained in $\mathcal{N}$ and whose sort equals the sort of N, and $N_0^o \in \mathcal{N}$ is a starting nonterminal, being of sort o. We assume that elements of $\mathcal{N}$ are not used as bound variables, and that $\mathcal{R}(N)$ is not a nonterminal. The order of the scheme is defined as the maximum of complexities of $\mathcal{R}(N)$ over all its nonterminals N.

The infinitary λ-term *generated by* a scheme $\mathcal{G} = (\mathcal{N}, \mathcal{R}, N_0^o)$, denoted $\Lambda(\mathcal{G})$, is defined as the limit of the following process starting from N_0^o: take any nonterminal N appearing in the current term, and replace it by $\mathcal{R}(N)$ (so that every nonterminal is eventually replaced). Observe that in the limit we obtain a closed λ-term of sort o and of complexity not greater than the order of the scheme. The *language of* $\mathcal{G}$ is defined as $\mathcal{L}(\mathcal{G}) = \mathcal{L}(BT(\Lambda(\mathcal{G})))$.

We remark that according to our definition all subterms of all λ-terms (and all nonterminals as well) have homogeneous sorts; usually it is not assumed that sorts used in a scheme are homogeneous. It is, however, the case that any scheme using also non-homogeneous sorts can be converted into one in which all sorts are homogeneous, and that this can be done in logarithmic space [20]. We make the homogeneity assumption for technical convenience.

A *word* is defined as a tree in which every node has at most one child (such a tree can be identified with a word understood in the classic sense). We say that a λ-term P is *word-recognizing* if for every its subterm of the form $a\langle P_1,\dots,P_r\rangle$ with $a\in\Sigma$ it holds $r\le 1$; a scheme $\mathcal{G}$ is *word-recognizing* if $\Lambda(\mathcal{G})$ is word-recognizing. In this case, all elements of $\mathcal{L}(BT(P))$ or $\mathcal{L}(\mathcal{G})$, respectively, are words.

▶ **Example 1.** Consider the higher-order recursion scheme $\mathcal{G}_1$ with two nonterminals, M^o (taken as the starting nonterminal) and $\mathsf{N}^{(o\to o)\to o}$, and with rules

$$\mathcal{R}(\mathsf{M}) = \mathsf{N}\,(\lambda\mathsf{x}.\mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle,\mathsf{b}\langle\mathsf{x}\rangle\rangle)\,, \qquad \mathcal{R}(\mathsf{N}) = \lambda\mathsf{f}.\mathsf{nd}\langle\mathsf{f}\,(\mathsf{c}\langle\rangle),\mathsf{N}\,(\lambda\mathsf{y}.\mathsf{f}\,(\mathsf{f}\,\mathsf{y}))\rangle\,.$$

We obtain $\Lambda(\mathcal{G}_1) = R_1\,(\lambda\mathsf{x}.\mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle,\mathsf{b}\langle\mathsf{x}\rangle\rangle)$, where R_1 is the unique λ-term such that $R_1 = \lambda\mathsf{f}.\mathsf{nd}\langle\mathsf{f}\,(\mathsf{c}\langle\rangle),R_1\,(\lambda\mathsf{y}.\mathsf{f}\,(\mathsf{f}\,\mathsf{y}))\rangle$. We have $BT(\Lambda(\mathcal{G}_1)) = \mathsf{nd}\langle T_{2^0},\mathsf{nd}\langle T_{2^1},\mathsf{nd}\langle T_{2^2},\dots\rangle\rangle\rangle$, where $T_0=\mathsf{c}\langle\rangle$ and $T_{i+1}=\mathsf{nd}\langle\mathsf{a}\langle T_i\rangle,\mathsf{b}\langle T_i\rangle\rangle$. In $\mathcal{L}(\mathcal{G}_1)$ we have words of length 2^i+1 for all $i\in\mathbb{N}$, where the first 2^i letters are chosen from $\{\mathsf{a},\mathsf{b}\}$ arbitrarily, and the last letter is c. In the following examples we will continue to consider this scheme, together with the set $A=\{\mathsf{a},\mathsf{b}\}$.

3 Type System for the Diagonal Problem

In this section we introduce a type system that allows to solve the diagonal problem for schemes.

▶ **Definition 2.** For a set of trees L and a set of symbols A, the predicate $Diag_A(L)$ holds if for every $n\in\mathbb{N}$ there is some $T\in L$ with at least n occurrences of every symbol from A. The *diagonal problem* for tree-recognizing order-m schemes is to decide whether $Diag_A(\mathcal{L}(\mathcal{G}))$ holds, given a scheme $\mathcal{G}$ of order at most m and a set A. The diagonal problem for *word-recognizing* order-m schemes is as the above, but with the restriction that $\mathcal{G}$ is word-recognizing.

▶ **Theorem 3.** *For $m\ge 1$, the diagonal problem for word-recognizing order-$(m+1)$ schemes is m-EXPTIME-complete. For $m\in\{-1,0\}$ it is NP-complete.*

Throughout the rest of this section we solve the diagonal problem for word-recognizing schemes, and thus all schemes considered here are assumed to be word-recognizing. Moreover, we fix a set of symbols A, for which we want to solve the diagonal problem.

Intuitions. The main novelty of our type system lies in labeling nodes of type derivations by two kinds of labels called flags and markers. To each marker we assign a number, called an order. Flags, beside of their order, are also identified by a symbol from A; thus we have (k,a)-flags for $k\in\mathbb{N}$ and $a\in A$. While deriving a type for a λ-term of complexity at most $m+1$, we use markers of order from the range $0,\dots,m$, and flags of order from the range $1,\dots,m+1$.

Let P_{m+1} be a λ-term of complexity at most $m+1$. Recall that our goal is to describe a word $T\in\mathcal{L}(BT(P_{m+1}))$ using a type derivation for P_{m+1} itself. While doing that, we want to preserve the information that T has many appearances of every symbol from A.

Since T can be found in some finite prefix of $BT(P_{m+1})$, in order to find T it is enough to perform finitely many β-reductions from P_{m+1}. Moreover, thanks to the fact that all sorts are homogeneous, the β-reductions can be rearranged so that those of higher order are performed first. Namely, we can find λ-terms $P_0,\dots,P_m$ such that

$$P_{m+1}\to^*_{\beta(m)} P_m \to^*_{\beta(m-1)} \cdots \to^*_{\beta(0)} P_0 \qquad\text{and}\qquad P_0\to^*_{\mathsf{nd}} T\,.$$

Some prefix of P_0 can be seen as a tree, in which we can find a path on which there are all symbols of T, and some additional nd symbols. Let us place an order-0 marker in the leaf ending this path. Additionally, for every symbol $a \in A$, we place $(1, a)$-flags in all a-labeled nodes of the considered path.

Next, we proceed back to P_1. The leaf constructor of P_0 containing our order-0 marker was created out of some particular appearance of such constructor in P_1; let us put there as well the order-0 marker. Similarly, we find node constructors in P_1 out of which in P_0 we obtain node constructors with flags, and we transfer the flags back to P_1. The crucial observation is that no two flagged node constructors of P_0 could come out of a single node constructor of P_1. Indeed, recall that all the β-reductions between P_1 and P_0 are of order 0. This means that in every such β-reduction we take a whole subtree (i.e., a λ-term of sort o) of P_1, and we replace it somewhere, possibly replicating it. But since all flags lie in P_0 on a single path, they may lie only in at most one copy of the replicated subtree. In effect, the number of appearances of order-1 flags of every kind is the same in P_1 as in P_0.

We cannot directly repeat the same reasoning to move flags from P_1 back to P_2, since now there is a problem: a single node constructor in P_2 may result in multiple (uncontrollably many) node constructors with a flag in P_1. We rescue ourselves by considering only $|A|$ paths in P_1, one for each symbol in A. Namely, for every symbol $a \in A$ we place in P_1 a marker of order 1, choosing in this way the path from the root to the position of this marker. Then, for every node labeled by a $(1, a)$-flag we place a $(2, a)$-flag in the closest ancestor that lies on the chosen path. Although the number of $(2, a)$-flags may be smaller than the number of $(1, a)$-flags (the closest ancestor on the path may be the same for multiple $(1, a)$-flags), we can ensure that it is smaller only logarithmically; to do so, we choose the marked node in a clever way: staring from the root, we always proceed to this subtree in which the number of $(1, a)$-flags is the largest. In effect, if the number of $(1, a)$-flags was "very large", then also the number of $(2, a)$-flags will be "very large".

Once for every $a \in A$ all $(2, a)$-flags lie on a single path of P_1, we can transfer them back to P_2 without changing their number. Then in P_2 we again reduce to $|A|$ paths by introducing markers of order 2, and so on. At the end we obtain some labeling of P_{m+1} by several kinds of flags and markers. The goal of the type system we develop is, roughly speaking, to ensure that a labeling of P_{m+1} actually is obtainable in the process as above (in fact, we will not be labeling nodes of P_{m+1} itself, but rather nodes of a type derivation for P_{m+1}).

Type Judgments. Recall that A is the set of symbols for which we want to solve the diagonal problem. For storing the information about flags and markers used in a derivation of a type we use flag sets and marker multisets. Recall that a flag is parameterized by a pair (k, a), where $k \in \mathbb{N}$ is called an order, and $a \in A$ is called a symbol. For flags it is enough to remember for every order whether at least one flag of this order was used, and if so, then also a symbol of this flag (if flags with multiple symbols were used, it is enough for us to remember just one of these symbols). Thus for $m \in \mathbb{N}$ we define $\mathcal{F}_m$ to contain sets $F \subseteq \{1, \dots, m\} \times A$ such that $(k, a), (k, b) \in F$ implies $a = b$. Such sets F are called *m-bounded flag sets.* For markers the situation is slightly different, as we want to remember precisely how many markers were used. Moreover, markers do not have a symbol, only an order. We thus define $\mathcal{M}_m$ to contain functions $M : \mathbb{N} \to \{0, \dots, |A|\}$ such that $M(0) \leq 1$ and $M(k) = 0$ for all $k > m$. Such functions M are called *m-bounded marker multisets.*

By $M + M'$ and $M - M'$ we mean the coordinatewise sum or difference, respectively. We use $\mathbf{0}$ to denote a function that maps every element of its domain to 0 (where the domain should be always clear from the context). By $\{\!|k_1, \dots, k_n|\!\}$ we mean the multiset M such that $M(k) = |\{i \in \{1, \dots, n\} \mid k_i = k\}|$ for all $k \in \mathbb{N}$. When $F \in \mathcal{F}_m$, $M \in \mathcal{M}_m$, $n \in \mathbb{N}$, and $\Box$ is one of $\leq$, $>$, we write $F{\restriction}_{\Box n}$ for $\{(k, a) \in F \mid k \Box n\}$, and $M{\restriction}_{\Box n}$ for the function that maps every k to $M(k)$ if $k \Box n$, and to 0 if $\neg(k \Box n)$.

Next, for every sort α and for $m \in \mathbb{N}$ we define three sets: the set $\mathcal{T}^\alpha$ of *types* of sort α, the set $\mathcal{TT}^\alpha_m$ of *m-bounded type triples* of sort α, and the set $\mathcal{TC}^\alpha$ of *triple containers* of sort α. They are defined by mutual induction on the structure of α.

If $\alpha = \alpha_1 \to \dots \to \alpha_s \to o$, the set $\mathcal{T}^\alpha$ contains types that are of the form $C_1 \to \dots \to C_s \to o$, where $C_i \in \mathcal{TC}^{\alpha_i}$ for $i \in \{1, \dots, s\}$.

Type triples in $\mathcal{TT}^\alpha_m$ are just triples $(F, M, C_1 \to \dots \to C_s \to o) \in \mathcal{F}_m \times \mathcal{M}_m \times \mathcal{T}^\alpha$, where $M(k) = 0$ for all $(k, a) \in F$, and where $M(0) + \sum_{i=1}^{s} \mathsf{Mk}(C_i)(0) = 1$ (we will define $\mathsf{Mk}(C_i)$ soon). These triples store a type, together with the information about flags and markers used while deriving this type. In order to distinguish type triples from types, the former are denoted by letters with a hat, like $\hat{\tau}$. We also define a function Mk that extracts the marker multiset out of a type triple: $\mathsf{Mk}(\hat{\tau}) = M$ for $\hat{\tau} = (F, M, \tau)$. A type triple is *balanced* if $\mathsf{Mk}(\hat{\tau}) = \mathbf{0}$; otherwise it is *unbalanced*.

Triple containers are used to store (multi)sets of type triples that have to be derived for an argument of a λ-term, or for a λ-term substituted for a free variable. For balanced type triples, triple containers behave like sets, that is, they remember only whether every balanced type triple is required or not. Conversely, for unbalanced type triples, triple containers behave like multisets, that is, they remember precisely how many times every unbalanced type triple is required. Thus, formally, in $\mathcal{TC}^\alpha$ we have functions $C\colon \mathcal{TT}^\alpha_{ord(\alpha)} \to \{0, \dots, |A|\}$ such that $C(\hat{\tau}) \leq 1$ if $\mathsf{Mk}(\hat{\tau}) = \mathbf{0}$. For $C \in \mathcal{TC}^\alpha$ we define $\mathsf{Mk}(C) = \sum_{\hat{\tau} \in \mathcal{TT}^\alpha_{ord(\alpha)}} \sum_{i=1}^{C(\hat{\tau})} \mathsf{Mk}(\hat{\tau})$. For two triple containers $C, D \in \mathcal{TC}^\alpha$ we define their sum $C \sqcup D\colon \mathcal{TT}^\alpha_{ord(\alpha)} \to \mathbb{N}$ so that for every $\hat{\tau} \in \mathcal{TT}^\alpha_{ord(\alpha)}$,

$$(C \sqcup D)(\hat{\tau}) = \begin{cases} C(\hat{\tau}) + D(\hat{\tau}) & \text{if } \mathsf{Mk}(\hat{\tau}) \neq \mathbf{0}\,, \\ \max(C(\hat{\tau}), D(\hat{\tau})) & \text{if } \mathsf{Mk}(\hat{\tau}) = \mathbf{0}\,. \end{cases}$$

We also say that $C \sqsubseteq D$ if $C(\hat{\tau}) = D(\hat{\tau})$ for every unbalanced $\hat{\tau} \in \mathcal{TT}^\alpha_{ord(\alpha)}$, and $C(\hat{\tau}) \leq D(\hat{\tau})$ for every balanced $\hat{\tau} \in \mathcal{TT}^\alpha_{ord(\alpha)}$. We sometimes write $\{\!|\hat{\tau}_1, \dots, \hat{\tau}_n|\!\}$ or $\{\!|\hat{\tau}_i \mid i \in \{1, \dots, n\}|\!\}$ to denote the triple container C such that $C(\hat{\sigma}) = |\{i \in \{1, \dots, n\} \mid \hat{\tau}_i = \hat{\sigma}\}|$ for every unbalanced type triple $\hat{\sigma}$, and $C(\hat{\sigma}) = 1 \Leftrightarrow \exists i \in \{1, \dots, n\} \,.\, \hat{\tau}_i = \hat{\sigma}$ for every balanced type triple $\hat{\sigma}$.

A *type environment* is a function Γ that maps every variable x^α to a triple container from $\mathcal{TC}^\alpha_{ord(\alpha)}$. We use ε to denote the type environment mapping every variable to $\mathbf{0}$. When $\Gamma(x) = \mathbf{0}$, by $\Gamma[x \mapsto C]$ we denote the type environment that maps x to C, and every other variable y to $\Gamma(y)$ (whenever we write $\Gamma[x \mapsto C]$, we implicitly require that $\Gamma(x) = \mathbf{0}$). For two type environments Γ, Γ' we define their sum $\Gamma \sqcup \Gamma'$ so that $(\Gamma \sqcup \Gamma')(x) = \Gamma(x) \sqcup \Gamma'(x)$ for every variable x.

A *type judgment* is of the form $\Gamma \vdash_m P : \hat{\tau} \triangleright c$, where Γ is a type environment, $m \in \mathbb{N}$ is called the *order* of the type judgment, P is a λ-term, $\hat{\tau}$ is an m-bounded type triple of the same sort as P (i.e. $\hat{\tau} \in \mathcal{TT}^\alpha_m$ when P is of sort α), and c is a function $A \to \mathbb{N}$ called a *flag counter*.

As usually for intersection types, the intuitive meaning of a type $C \to \tau$ is that a λ-term having this type can return a λ-term having type τ, while taking an argument for which we can derive all type triples from C. Let us now explain the meaning of a type judgment $\Gamma \vdash_m P : (F, M, \tau) \triangleright c$. Obviously τ is the type derived for P, and Γ contains type triples that could be used for free variables of P in the derivation. As explained above for triple containers, balanced and unbalanced type triples behave differently: all unbalanced type triples assigned to variables by Γ have to be used exactly once in the derivation; conversely, balanced type triples may be used any number of times. Going further, the order m of the type judgment bounds the order of flags and markers that can be used in the derivation: flags can be of order at most $m+1$, and markers of order at most m. The multiset M counts markers used in the derivation, together with those provided by free variables (i.e., we imagine that some derivations, specified by the type environment, are already substituted in our derivation for free variables); we, however, do not include markers provided by arguments of the λ-term (i.e. coming from the triple containers C_i when $\tau = C_1 \to \ldots \to C_s \to o$). The set F contains an information about flags of order at most m used in the derivation. A pair (k, a) can be contained in F if a (k, a)-flag is placed in the derivation itself, or provided by a free variable, or provided by an argument. We do not have to keep in F all such pairs, i.e., if we can derive a type triple with some flag set F, then we can derive it also with every subset of F as the flag set. In fact, we cannot keep in F all such pairs due to two restrictions. First, the definition of a flag set allows to have in F at most one pair (k, a) for every order k. Second, we intentionally remove from F all pairs (k, a) for which $M(k) > 0$. Finally, in a type judgment we have a function c, called a *flag counter*, that for each symbol a counts the number of $(m+1, a)$-flags present in the derivation.

Type System. Before giving rules of the type system, let us state two general facts. First, all type derivations are assumed to be finite – although we derive types mostly for infinite λ-terms, each type derivation analyzes only a finite part of a term. Second, we require that premisses and conclusions of all rules are valid type judgments. For example, when the type environment appearing in the conclusion of a rule is $\Gamma \sqcup \Gamma'$, this implies that for all x and all unbalanced type triples $\hat{\tau}$ it holds $\Gamma(x)(\hat{\tau}) + \Gamma'(x)(\hat{\tau}) \leq |A|$ (so that $(\Gamma \sqcup \Gamma')(x)$ is indeed a valid triple container). Let us also remark that rules of the type system will guarantee that the order m of all type judgments used in a derivation will be the same.

Rules of the type system correspond to particular constructs of λ-calculus. We start by giving the first three rules:

$$\frac{M{\restriction_{\leq ord(x)}} = M'}{\varepsilon[x \mapsto \{\!|(F, M', \tau)|\!\}] \vdash_m x : (F, M, \tau) \triangleright \mathbf{0}}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash_m P_i : \hat{\tau} \triangleright c \qquad i \in \{1, \ldots, r\}}{\Gamma \vdash_m \mathsf{nd}\langle P_1, \ldots, P_r\rangle : \hat{\tau} \triangleright c}\ (\textsc{Nd})$$

$$\frac{\Gamma[x \mapsto C'] \vdash_m P : (F, M, \tau) \triangleright c \qquad C' \sqsubseteq C}{\Gamma \vdash_m \lambda x.P : (F, M - \mathsf{Mk}(C), C \to \tau) \triangleright c}\ (\lambda)$$

The ($\textsc{Var}$) rule allows to have in the resulting marker multiset M some numbers that do not come from the multiset assigned to x by the type environment; these are the orders of markers placed in the leaf using this rule. Notice, however, that we allow here only orders greater than $ord(x)$. This is consistent with the intuitive description of the type system (page 4), which says that a marker of order k can be put in a place that will be a leaf after performing all β-reductions of order at least k. Indeed, the variable x remains a leaf after performing β-reductions of orders greater than $ord(x)$, but while performing β-reductions of order $ord(x)$ this leaf will be replaced by a subterm substituted for x. Recall also that, by definition of a type judgment, we require that $(F, M', \tau) \in \mathcal{TT}^{\alpha}_{ord(x)}$ and $(F, M, \tau) \in \mathcal{TT}^{\alpha}_{m}$, for appropriate sort α; this introduces a bound on maximal numbers that may appear in F and M.

▶ **Example 4.** Denoting $\hat{\rho}_0 = (\emptyset, \{\!|0|\!\}, o)$ we can derive:

$$\frac{}{\varepsilon[\mathsf{x} \mapsto \{\!|\hat{\rho}_0|\!\}] \vdash_1 \mathsf{x} : (\emptyset, \{\!|0|\!\}, o) \triangleright \mathbf{0}}\ (\textsc{Var}) \qquad \frac{}{\varepsilon[\mathsf{x} \mapsto \{\!|\hat{\rho}_0|\!\}] \vdash_1 \mathsf{x} : (\emptyset, \{\!|0, 1, 1|\!\}, o) \triangleright \mathbf{0}}\ (\textsc{Var})$$

In the derivation on the right, two markers of order 1 are placed in the conclusion of the rule.

We see that to derive a type for the nondeterministic choice $\mathsf{nd}\langle P_1, \dots, P_r\rangle$, we need to derive it for one of the subterms $P_1, \dots, P_r$.

For the (λ) rule, recall that $C' \sqsubseteq C$ means that in C' we have all unbalanced type triples from C, and some subset of balanced type triples from C. Thus in a subderivation concerning the λ-term P, we need to use all unbalanced type triples provided by an argument of $\lambda x.P$, while balanced type triples may be used or not. Recall also that we intend to store in the marker multiset the markers contained in the derivation itself and those provided by free variables, but not those provided by arguments. Because of this, in the conclusion of the rule we remove from M the markers provided by x. It is required, implicitly, that the result remains nonnegative. The set F, unlike M, stores also flags provided by arguments, so we do not need to remove anything from F.

▶ **Example 5.** In this example we show how the (ND) and (λ) rules can be used. Notice that in the conclusion of the (λ) rule, in both derivations, we remove 0 from the marker multiset, because an order-0 marker is provided by x.

$$\dfrac{\dfrac{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{a}\langle\mathsf{x}\rangle : (\{(1,\mathsf{a})\}, \{\!|0|\!\}, o) \triangleright \mathbf{0}}{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle, \mathsf{b}\langle\mathsf{x}\rangle\rangle : (\{(1,\mathsf{a})\}, \{\!|0|\!\}, o) \triangleright \mathbf{0}}\,(\text{ND})}{\varepsilon \vdash_1 \lambda\mathsf{x}.\mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle, \mathsf{b}\langle\mathsf{x}\rangle\rangle : (\{(1,\mathsf{a})\}, \mathbf{0}, \{\!|\hat\rho_0|\!\} \to o) \triangleright \mathbf{0}}\,(\lambda)$$

$$\dfrac{\dfrac{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{a}\langle\mathsf{x}\rangle : (\emptyset, \{\!|0,1,1|\!\}, o) \triangleright \{(\mathsf{a},1),(\mathsf{b},0)\}}{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle, \mathsf{b}\langle\mathsf{x}\rangle\rangle : (\emptyset, \{\!|0,1,1|\!\}, o) \triangleright \{(\mathsf{a},1),(\mathsf{b},0)\}}\,(\text{ND})}{\varepsilon \vdash_1 \lambda\mathsf{x}.\mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle, \mathsf{b}\langle\mathsf{x}\rangle\rangle : (\emptyset, \{\!|1,1|\!\}, \{\!|\hat\rho_0|\!\} \to o) \triangleright \{(\mathsf{a},1),(\mathsf{b},0)\}}\,(\lambda)$$

The next three rules use a predicate Comp_m, saying how flags and markers from premisses contribute to the conclusion. It takes "as input" pairs (F_i, c_i) for $i \in I$, consisting of a flag set F_i and a flag counter c_i from some premiss. Moreover, the predicate takes a marker multiset M that will appear in the conclusion of the rule. The goal is to compute a flag set F and a flag counter c that should be placed in the conclusion. First, for each $k \in \{1, \dots, m+1\}$ consecutively, we decide which flags of order k should be placed in the considered node of a type derivation. We follow here the rules mentioned in the intuitive description. Namely, we place a (k,a)-flag if we are on the path leading to a marker of order $k-1$ (i.e., if $M(k-1) > 0$), and simultaneously we receive an information about a $(k-1,a)$-flag. By receiving this information we mean that either a $(k-1,a)$-flag was placed in the current node, or $(k-1,a)$ belongs to some set F_i. Actually, we place multiple (k,a)-flags: one per each $(k-1,a)$-flag placed in the current node, and one per each set F_i containing $(k-1,a)$. Then, we compute F and c. In $c(a)$, for every $a \in A$, we store the number of $(m+1,a)$-flags: we sum all the flag counters c_i, and we add the number of $(m+1,a)$-flags placed in the current node. In F, we allow to keep elements of all F_i, and we allow to add pairs (k,a) for flags that were placed in the current node, but it can be chosen "nondeterministically" which of them are actually taken to F, and which are dropped. It is often necessary to drop some elements, since when the set F is used in a type triple, the definitions of a flag set and of a type triple put additional requirements on this set.

Below we give a formal definition, in which $f'_{k,a}$ contains the number of (k,a)-flags placed in the current node, while $f_{k,a}$ additionally counts the number of premisses for which $(k,a) \in F_i$. We say that $(F,c) \in \mathit{Comp}_m(M; ((F_i,c_i))_{i\in I})$ when

$$F \subseteq \{(k,a) \mid f_{k,a} > 0\}, \qquad \text{and} \qquad c(a) = f_{m+1,a} + \sum_{i\in I} c_i(a) \qquad \text{for all } a \in A,$$

where, for $k \in \{0, \dots, m+1\}$ and $a \in A$,

$$f_{k,a} = f'_{k,a} + \sum_{i \in I} |F_i \cap \{(k,a)\}|, \qquad f'_{k,a} = \begin{cases} 0 & \text{if } k = 0 \text{ or } M(k-1) = 0, \\ f_{k-1,a} & \text{otherwise.} \end{cases}$$

We now present rules for node constructors using symbols other than nd:

$$\frac{(F,c) \in \mathit{Comp}_m(M; (\{(0,a)\}, \mathbf{0})) \qquad a \neq \mathsf{nd}}{\varepsilon \vdash_m a\langle\rangle : (F, M, o) \triangleright c} \ (\textsc{Con0})$$

$$\frac{\Gamma \vdash_m P : (F', M, o) \triangleright c' \qquad (F,c) \in \mathit{Comp}_m(M; (\{(0,a)\}, \mathbf{0}), (F', c')) \qquad a \neq \mathsf{nd}}{\Gamma \vdash_m a\langle P\rangle : (F, M, o) \triangleright c} \ (\textsc{Con1})$$

In these rules we do not claim that the set $\{(0,a)\}$ passed to Comp_m is an element of $\mathcal{F}_m$ (and in fact it is not, because the order is 0, which is forbidden for flags; we also do not necessarily have that $a \in A$). The effect of passing this set is that if $M(0) > 0$ (i.e., we are on the path to the order-0 marker) and $a \in A$, then Comp_m places a $(1,a)$-flag in the current node, and maybe also some (k,a)-flags for higher k. In the (Con0) rule, i.e., if we are in a leaf, we are allowed to place markers of arbitrary order: the marker multiset M may be arbitrary.

▶ **Example 6.** The (Con1) rule may be instantiated in the following ways:

$$\frac{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{x} : (\emptyset, \{\!|0|\!\}, o) \triangleright \mathbf{0}}{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{a}\langle\mathsf{x}\rangle : (\{(1,\mathsf{a})\}, \{\!|0|\!\}, o) \triangleright \mathbf{0}} \ (\textsc{Con1})$$

$$\frac{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{x} : (\emptyset, \{\!|0,1,1|\!\}, o) \triangleright \mathbf{0}}{\varepsilon[\mathsf{x} \mapsto \{\!|\hat\rho_0|\!\}] \vdash_1 \mathsf{a}\langle\mathsf{x}\rangle : (\emptyset, \{\!|0,1,1|\!\}, o) \triangleright \{(\mathsf{a},1), (\mathsf{b},0)\}} \ (\textsc{Con1})$$

In the first example, a $(1,\mathsf{a})$-flag is placed in the conclusion of the rule (because the marker multiset contains 0, the pair $(0,\mathsf{a})$ passed to the Comp_1 predicate results in the $(1,\mathsf{a})$ pair in the flag set). In the second example, $(1,\mathsf{a})$- and $(2,\mathsf{a})$-flags are placed in the conclusion of the (Con1) rule: since order-1 markers are visible, we do not put $(1,\mathsf{a})$ to the flag set, but instead we create a $(2,\mathsf{a})$-flag, which results in increasing the flag counter.

The last rule describes application:

$$\frac{\begin{array}{c}\Gamma' \vdash_m P : (F', M', \{\!|(F_i{\restriction_{\leq \mathit{ord}(Q)}}, M_i{\restriction_{\leq \mathit{ord}(Q)}}, \tau_i) \mid i \in I|\!\} \to \tau) \triangleright c' \\ \Gamma_i \vdash_m Q : (F_i, M_i, \tau_i) \triangleright c_i \text{ for each } i \in I \qquad M = M' + \sum_{i \in I} M_i \qquad \mathit{ord}(Q) \leq m \\ (F,c) \in \mathit{Comp}_m(M; (F', c'), ((F_i{\restriction_{> \mathit{ord}(Q)}}, c_i))_{i \in I}) \qquad \{(k,a) \in F' \mid M(k) = 0\} \subseteq F\end{array}}{\Gamma' \sqcup \bigsqcup_{i \in I} \Gamma_i \vdash_m P\,Q : (F, M, \tau) \triangleright c} \ (@)$$

In this rule, it is allowed (and potentially useful) that for two different $i \in I$ the type triples (F_i, M_i, τ_i) are equal. It is also allowed that $I = \emptyset$, in which case no type needs to be derived for Q. Observe how flags and markers coming from premisses concerning Q are propagated: only flags and markers of order $k \leq \mathit{ord}(Q)$ are visible to P, while only flags of order $k > \mathit{ord}(Q)$ are passed to the Comp_m predicate. This can be justified if we recall the intuitions staying behind the type system (see page 4). Indeed, while considering flags and markers of order k, we should imagine the λ-term obtained from the current λ-term by performing all β-reductions of order at least k; the distribution of flags and markers of order

k in the current λ-term actually simulates their distribution in this imaginary λ-term. Thus, if $k \leq \mathit{ord}(Q)$, then our application will disappear in this imaginary λ-term (thanks to the homogenity assumption), and Q will be already substituted somewhere in P; for this reason we need to pass the information about flags and markers of order k from Q to P. Conversely, if $k > \mathit{ord}(Q)$, then in the imaginary λ-term the considered application will be still present, and in consequence the subterm corresponding to P will not see flags and markers of order k placed in the subterm corresponding to Q. The condition $\{(k,a) \in F' \mid M(k) = 0\} \subseteq F$ (saying that some flags from F' cannot disappear) is useful for proofs.

▶ **Example 7.** Recalling that $\hat{\rho}_0 = (\emptyset, \{\!|0|\!\}, o)$, denote by $\hat{\tau}_{\mathsf{a}}$ and $\hat{\tau}_{\mathsf{m}}$ the type triples derived in Example 5: $\hat{\tau}_{\mathsf{a}} = (\{(1,\mathsf{a})\}, \mathbf{0}, \{\!|\hat{\rho}_0|\!\} \to o)$ and $\hat{\tau}_{\mathsf{m}} = (\emptyset, \{\!|1,1|\!\}, \{\!|\hat{\rho}_0|\!\} \to o)$. We can derive:

$$\dfrac{\dfrac{\dfrac{}{\varepsilon[\mathsf{f} \mapsto \{\!|\hat{\tau}_{\mathsf{a}}|\!\}] \vdash_1 \mathsf{f} : \hat{\tau}_{\mathsf{a}} \triangleright \mathbf{0}} \qquad \dfrac{\dfrac{}{\varepsilon[\mathsf{f} \mapsto \{\!|\hat{\tau}_{\mathsf{m}}|\!\}] \vdash_1 \mathsf{f} : \hat{\tau}_{\mathsf{m}} \triangleright \mathbf{0}} \qquad \dfrac{}{\varepsilon[\mathsf{y} \mapsto \{\!|\hat{\rho}_0|\!\}] \vdash_1 \mathsf{y} : \hat{\rho}_0 \triangleright \mathbf{0}}}{\varepsilon[\mathsf{f} \mapsto \{\!|\hat{\tau}_{\mathsf{m}}|\!\}, \mathsf{y} \mapsto \{\!|\hat{\rho}_0|\!\}] \vdash_1 \mathsf{f}\,\mathsf{y} : (\emptyset, \{\!|0,1,1|\!\}, o) \triangleright \mathbf{0}}\,(@)}{\varepsilon[\mathsf{f} \mapsto \{\!|\hat{\tau}_{\mathsf{a}}, \hat{\tau}_{\mathsf{m}}|\!\}, \mathsf{y} \mapsto \{\!|\hat{\rho}_0|\!\}] \vdash_1 \mathsf{f}\,(\mathsf{f}\,\mathsf{y}) : (\emptyset, \{\!|0,1,1|\!\}, o) \triangleright \{(\mathsf{a},1),(\mathsf{b},0)\}}\,(@)}{\varepsilon[\mathsf{f} \mapsto \{\!|\hat{\tau}_{\mathsf{a}}, \hat{\tau}_{\mathsf{m}}|\!\}] \vdash_1 \lambda\mathsf{y}.\mathsf{f}\,(\mathsf{f}\,\mathsf{y}) : \hat{\tau}_{\mathsf{m}} \triangleright \{(\mathsf{a},1),(\mathsf{b},0)\}}\,(\lambda)$$

Below the lower (@) rule the information about a $(1,\mathsf{a})$-flag (from the first premiss) meets the information about a marker of order 1 (from the second premiss), and thus a $(2,\mathsf{a})$-flag is placed, which increases the flag counter.

Denote $\hat{\rho}_m = (\emptyset, M_m^{all}, o)$, where $M_m^{all} \in \mathcal{M}_m$ is such that $M_m^{all}(0) = 1$ and $M_m^{all}(k) = |A|$ for all $k \in \{1, \dots, m\}$. The key property of the type system is described by the following theorem.

▶ **Theorem 8.** *Let $m \in \mathbb{N}$, and let P be a closed word-recognizing λ-term of sort o and complexity at most $m+1$. Then $\mathit{Diag}_A(\mathcal{L}(BT(P)))$ holds if and only if for every $n \in \mathbb{N}$ we can derive $\varepsilon \vdash_m P : \hat{\rho}_m \triangleright c_n$ with some c_n such that $c_n(a) \geq n$ for all $a \in A$.*

We omit the proof of this theorem. The overall idea is to follow the intuitions described on page 4, and consider only such sequences of β-reductions in which reductions of higher orders are performed before β-reductions of lower orders. The details are tedious, but rather standard. Actually, a quite similar proof was performed in our recent work [19] concerning the single-letter case.

▶ **Example 9.** Denote $\hat{\sigma}_R = (\emptyset, \{\!|0|\!\}, \{\!|\hat{\tau}_{\mathsf{a}}, \hat{\tau}_{\mathsf{b}}, \hat{\tau}_{\mathsf{m}}|\!\} \to o)$, where $\hat{\tau}_{\mathsf{b}} = (\{(1,\mathsf{b})\}, \mathbf{0}, \{\!|\hat{\rho}_0|\!\} \to o)$. We can derive $\varepsilon \vdash_1 R_1 : \hat{\sigma}_R \triangleright \mathbf{0}$ by descending to the first child of the outermost $\mathsf{nd}\langle\cdot,\cdot\rangle$ in $R_1 = \lambda\mathsf{f}.\mathsf{nd}\langle\mathsf{f}\,(\mathsf{c}\langle\rangle), R_1\,(\lambda\mathsf{y}.\mathsf{f}\,(\mathsf{f}\,\mathsf{y}))\rangle$. Then, basing on a type judgment $\varepsilon \vdash_1 R_1 : \hat{\sigma}_R \triangleright c$ we can derive $\varepsilon \vdash_1 R_1 : \hat{\sigma}_R \triangleright \{(\mathsf{a}, c(\mathsf{a})+1), (\mathsf{b}, c(\mathsf{b}))\}$ using in particular the derivation fragment from Example 7, and similarly $\varepsilon \vdash_1 R_1 : \hat{\sigma}_R \triangleright \{(\mathsf{a}, c(\mathsf{a})), (\mathsf{b}, c(\mathsf{b})+1)\}$. By composing the above derivation fragments, we can derive $\varepsilon \vdash_1 R_1 : \hat{\sigma}_R \triangleright c$ for c that is arbitrarily large on both coordinates. Examples 4-6 contain derivations of type triples $\hat{\tau}_{\mathsf{a}}$ and $\hat{\tau}_{\mathsf{m}}$ for the λ-term $\lambda\mathsf{x}.\mathsf{nd}\langle\mathsf{a}\langle\mathsf{x}\rangle, \mathsf{b}\langle\mathsf{x}\rangle\rangle$; similarly we can derive the type triple $\hat{\tau}_{\mathsf{b}}$. Using the (@) rule one more time, we can derive $\varepsilon \vdash_1 \Lambda(\mathcal{G}_1) : \hat{\rho}_1 \triangleright c$ for c that is arbitrarily large on both coordinates.

▶ **Example 10.** Consider the scheme $\mathcal{G}_2$ obtained from $\mathcal{G}_1$ by changing the rule $\mathcal{R}(\mathsf{N})$ to $\lambda\mathsf{f}.\mathsf{nd}\langle\mathsf{f}\,(\mathsf{c}\langle\rangle), \mathsf{N}\,(\lambda\mathsf{y}.\mathsf{f}\,\mathsf{y})\rangle$. Then while deriving a type for $\lambda\mathsf{y}.\mathsf{f}\,\mathsf{y}$ we can use only one type triple: either $\hat{\tau}_{\mathsf{a}}$, or $\hat{\tau}_{\mathsf{b}}$, or $\hat{\tau}_{\mathsf{m}}$, which causes that the flag counter is not increased. Thus, by adopting the derivation fragment considered in the previous example, out of $\varepsilon \vdash_1 R_2 : \hat{\sigma}_R \triangleright c$ we can only derive $\varepsilon \vdash_1 R_2 : \hat{\sigma}_R \triangleright c$, with the same flag counter (where R_2 is defined analogously to R_1). Altogether, we can derive $\varepsilon \vdash_1 \Lambda(\mathcal{G}_2) : \hat{\rho}_1 \triangleright c$ only for c with $c(\mathsf{a}) + c(\mathsf{b}) \leq 1$. This corresponds to the fact that $\mathcal{L}(\mathcal{G}_2)$ contains only words with at most one letter from $\{\mathsf{a}, \mathsf{b}\}$.

▶ **Example 11.** In the derivation from Example 9 both order-1 markers were placed in the same leaf, corresponding to the subterm x. Consider, however, a scheme $\mathcal{G}_3$, where additionally to M and N we have a nonterminal $\mathsf{M_b}$ of sort o, and the rules are changed to:

$$\mathcal{R}(\mathsf{M}) = \mathsf{N}\,(\lambda \mathsf{x}.\mathsf{a}\langle \mathsf{x}\rangle)\,, \qquad \mathcal{R}(\mathsf{M_b}) = \mathsf{nd}\langle \mathsf{c}\langle\rangle, \mathsf{b}\langle \mathsf{M_b}\rangle\rangle\,,$$
$$\mathcal{R}(\mathsf{N}) = \lambda \mathsf{f}.\mathsf{nd}\langle \mathsf{f}\,\mathsf{M_b}, \mathsf{N}\,(\lambda \mathsf{y}.\mathsf{f}\,(\mathsf{f}\,\mathsf{y}))\rangle\,.$$

Here we need to place one order-1 marker (responsible for counting appearances of the a symbol) in a leaf corresponding to x, and another order-1 marker (responsible for counting appearances of the b symbol) in a leaf corresponding to $\mathsf{c}\langle\rangle$.

Effectiveness. We now justify how Theorem 3 follows from Theorem 8, i.e., how given a word-recognizing scheme $\mathcal{G}$ of order at most $m+1$ and a set of symbols A, one can check in m-EXPTIME whether $\varepsilon \vdash_m \Lambda(\mathcal{G}) : \hat{\rho}_m \triangleright c$ can be derived for flag counters c that are arbitrarily large on every coordinate. Let us say that two type judgments are equivalent if they differ only in values of the flag counter. One can see that if c is required to be large enough on every coordinate, then in the derivation of $\varepsilon \vdash_m \Lambda(\mathcal{G}) : \hat{\rho}_m \triangleright c$, for every symbol $a \in A$, there are two equivalent type judgments lying on the same path (the path may depend on a) and such that the a-coordinate of their flag counter differs. A derivation having this property will be called *pumpable*. This name is justified, because the opposite implication also holds: if we have a pumpable type derivation, then we can repeat (as many times as we want) its fragment between all these pairs of equivalent type judgments, increasing arbitrarily all coordinates of the flag counter in the resulting type judgment. This holds thanks to the following additivity property of our type system: if out of $\Gamma \vdash_m P : \hat{\tau} \triangleright c$ we can derive $\Gamma' \vdash_m P' : \hat{\tau}' \triangleright c'$, then out of $\Gamma \vdash_m P : \hat{\tau} \triangleright d$ we can derive $\Gamma' \vdash_m P' : \hat{\tau}' \triangleright c' + d - c$.

We exploit the above equivalence, and we also observe that while deriving $\varepsilon \vdash_m \Lambda(\mathcal{G}) : \hat{\rho}_m \triangleright c$ we may only use type judgments $\Gamma \vdash_m Q : \hat{\tau} \triangleright d$ (call them *useful*) in which Q is a subterm of $\Lambda(\mathcal{G})$ and $\Gamma(x) \neq \mathbf{0}$ only for variables x that are free in Q. It is not difficult to give an algorithm that checks whether there is a pumpable derivation of $\varepsilon \vdash_m \Lambda(\mathcal{G}) : \hat{\rho}_m \triangleright c$ (for some c), and works in time polynomial in the number of equivalence classes of useful type judgments (and exponential in $|A|$).

It remains to bound the number of these equivalence classes. We first bound the number of type triples. Essentially, type triples in $\mathcal{TT}_k^\alpha$ with $\alpha = \alpha_1 \to \dots \to \alpha_s \to o$ store “sets” of type triples from $\mathcal{TT}_{ord(\alpha_i)}^{\alpha_i}$, and thus their number grows exponentially when we increase the order of α by one. There is a slight exception for α of order 1: the number of type triples in $\mathcal{TT}_k^\alpha$ for such α is polynomial, not exponential. The reason is that, for a type $(C_1 \to \dots \to C_s \to o) \in \mathcal{T}_k^\alpha$ with $ord(\alpha) = 1$, the triple containers $C_1, \dots, C_s$ can contain altogether only at most one type triple (by definition it is required that $\sum_{i=1}^{s} \mathsf{Mk}(C_i)(0) \leq 1$ while, on the other hand, for type triples $\hat{\sigma} \in \mathcal{TT}_0^o$ it is required that $\mathsf{Mk}(\hat{\sigma})(0) = 1$). In effect, $|\mathcal{TT}_m^\alpha|$ for $ord(\alpha) \leq m+1$ is m-fold exponential in the size of $\mathcal{G}$. It easily follows that the number of equivalence classes of useful type judgments is also m-fold exponential in the size of $\mathcal{G}$, because all variables appearing in $\Lambda(\mathcal{G})$ are of order strictly smaller than $m+1$.

A trivial reduction from the problem of emptiness of $\mathcal{L}(\mathcal{G})$ shows that our problem is indeed m-EXPTIME-hard [15]. For $m \in \{-1, 0\}$ one can prove NP-completeness.

4 Extensions

Downward Closure. The downward closure of a language of words L, denoted $L\!\downarrow$, is the set of all scattered subwords (subsequences) of words from L. Recall that the downward closure of any set is always a regular language; moreover, it is a finite union of *ideals*, i.e., languages of the form $Y_0^*\{x_1,\varepsilon\}Y_1^*\dots\{x_n,\varepsilon\}Y_n^*$, where $x_1,\dots,x_n$ are letters, and $Y_0,\dots,Y_n$ are sets of letters. The main interest on the diagonal problem comes from the fact that this problem is closely related to computability of the downward closure of a language of words (where we aim in presenting the results by a list of ideals, or by a finite automaton). Indeed, having a word-recognizing scheme $\mathcal{G}$, it is not difficult to compute $\mathcal{L}(\mathcal{G})\!\downarrow$ by performing multiple calls to a procedure solving the diagonal problem (for products of $\mathcal{G}$ and some finite automata). The complexity of this algorithm is directly related to the size of its output. We, however, do not know any upper bound on the size of (a representation of) $\mathcal{L}(\mathcal{G})\!\downarrow$. A recently developed pumping lemma for nondeterministic schemes [2] may shed some new light on this subject (while pumping lemmata for deterministic schemes [10, 12] seem irrelevant here).

Instead of actually computing the downward closure, Zetzsche [24] proposed to consider the following decision problem of downward-closure inclusion: given two word-recognizing schemes $\mathcal{G},\mathcal{H}$ of order at most m, check whether $\mathcal{L}(\mathcal{G})\!\downarrow\ \subseteq \mathcal{L}(\mathcal{H})\!\downarrow$; he proved that this problem is co-m-NEXPTIME-hard. It would be interesting to give some upper bound on the complexity of this problem. Although, again, we do know how to do this, we can at least give a partial result.

▶ **Theorem 12.** *Let $m \geq 1$. Given a word-recognizing scheme $\mathcal{H}$ of order at most $m+1$, and an ideal I, the problem of deciding whether $I \subseteq \mathcal{L}(\mathcal{H})\!\downarrow$ is m-EXPTIME-complete.*

This is an easy consequence of Theorem 3: it is enough to appropriately combine $\mathcal{H}$ with I, and then solve the diagonal problem.

Tree-Generating Schemes. Although the main interest on the diagonal problem is for word-recognizing schemes, the problem can be also considered for tree-recognizing schemes. Let us see how our methods can be adopted to this more general case. Consider a tree $T \in \mathcal{L}(\mathcal{G})$, and a term P_0 such that $\Lambda(\mathcal{G}) \to_\beta^* P_0$ and $P_0 \to_{\mathsf{nd}}^* T$, i.e., that T can be found in a prefix of P_0. In the word case, we were placing order-1 flags in node constructors of T, and then we continued using the fact that they are all aligned along one path (as actually T consisted of a single path). This is no longer possible in the tree case. In order to resolve this issue, we additionally use flags of order 0, and we place them in node constructors of T (dispersed on multiple paths). Then, we choose only $|A|$ paths, by placing order-1 markers in $|A|$ leaves of P_0, and for every node labeled by a $(0,a)$-flag we place a $(1,a)$-flag in the closest ancestor that lies on a chosen path. In effect all order-1 flags are concentrated on only $|A|$ paths, and we can continue as in the word case. The described modification causes an exponential growth of the number of types, which results in the following theorem.

▶ **Theorem 13.** *For $m \geq 1$, the diagonal problem for tree-recognizing order-m schemes is m-EXPTIME-complete. For $m = 0$ it is NP-complete.*

Downward Closure for Trees. One can also consider the downward closure of a language of trees, defined as a set of all trees that can be homeomorphically embedded in trees from the language. By Kruskal's tree theorem [17] downward closures of tree languages are regular languages of trees. We notice, however, that (unlike for words) an algorithm solving the diagonal problem is highly insufficient for the purpose of computing the downward closure. Even in the single-letter case, in order to compute $L\!\downarrow$, one has to check, in particular, whether for every $n \in \mathbb{N}$, a full binary tree of depth n can be embedded in some tree from L; using the diagonal problem, we can only determine whether L contains arbitrarily large trees. Extending our techniques to this kind of problems is an interesting direction for further work.

References

1 Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968. doi:10.1145/321479.321488.

2 Kazuyuki Asada and Naoki Kobayashi. Pumping lemma for higher-order languages. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 97:1–97:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.97.

3 Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-push-down languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996. doi:10.1142/S0129054196000191.

4 Christopher H. Broadbent and Naoki Kobayashi. Saturation-based model checking of higher-order recursion schemes. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/LIPIcs.CSL.2013.129.

5 Lorenzo Clemente, Pawel Parys, Sylvain Salvati, and Igor Walukiewicz. Ordered tree-pushdown systems. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, volume 45 of *LIPIcs*, pages 163–177. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.163.

6 Lorenzo Clemente, Pawel Parys, Sylvain Salvati, and Igor Walukiewicz. The diagonal problem for higher-order recursion schemes is decidable. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 96–105. ACM, 2016. doi:10.1145/2933575.2934527.

7 Wojciech Czerwinski, Wim Martens, Lorijn van Rooijen, and Marc Zeitoun. A note on decidable separability by piecewise testable languages. In Adrian Kosowski and Igor Walukiewicz, editors, *Fundamentals of Computation Theory - 20th International Symposium, FCT 2015, Gdańsk, Poland, August 17-19, 2015, Proceedings*, volume 9210 of *Lecture Notes in Computer Science*, pages 173–185. Springer, 2015. doi:10.1007/978-3-319-22177-9_14.

8 Werner Damm. The IO- and oi-hierarchies. *Theor. Comput. Sci.*, 20:95–207, 1982. doi:10.1016/0304-3975(82)90009-3.

9 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 452–461. IEEE Computer Society, 2008. doi:10.1109/LICS.2008.34.

10 Alexander Kartzow and Pawel Parys. Strictness of the collapsible pushdown hierarchy. In Branislav Rovan, Vladimiro Sassone, and Peter Widmayer, editors, *Mathematical Foundations of Computer Science 2012 - 37th International Symposium, MFCS 2012, Bratislava, Slovakia, August 27-31, 2012. Proceedings*, volume 7464 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2012. doi:10.1007/978-3-642-32589-2_50.

11 Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428. ACM, 2009. doi:10.1145/1480881.1480933.

12 Naoki Kobayashi. Pumping by typing. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 398–407. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.46.

13 Naoki Kobayashi, Kazuhiro Inaba, and Takeshi Tsukada. Unsafe order-2 tree languages are context-sensitive. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8412 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2014. doi:10.1007/978-3-642-54830-7_10.

14 Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 179–188. IEEE Computer Society, 2009. doi:10.1109/LICS.2009.29.

15 Naoki Kobayashi and C.-H. Luke Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011. doi:10.2168/LMCS-7(4:9)2011.

16 Gregory M. Kobele and Sylvain Salvati. The IO and OI hierarchies revisited. *Inf. Comput.*, 243:205–221, 2015. doi:10.1016/j.ic.2014.12.015.

17 Joseph. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960. doi:10.2307/1993287.

18 Pawel Parys. A characterization of lambda-terms transforming numerals. *J. Funct. Program.*, 26:e12, 2016. doi:10.1017/S0956796816000113.

19 Pawel Parys. Intersection types and counting. In Naoki Kobayashi, editor, *Proceedings Eighth Workshop on Intersection Types and Related Systems, ITRS 2016, Porto, Portugal, 26th June 2016.*, volume 242 of *EPTCS*, pages 48–63, 2016. doi:10.4204/EPTCS.242.6.

20 Paweł Parys. Homogeneity without loss of generality. Submitted, 2017.

21 Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 61–72. ACM, 2014. doi:10.1145/2535838.2535873.

22 Sylvain Salvati and Igor Walukiewicz. Simply typed fixpoint calculus and collapsible pushdown automata. *Mathematical Structures in Computer Science*, 26(7):1304–1350, 2016. doi:10.1017/S0960129514000590.

23 Georg Zetzsche. An approach to computing downward closures. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2015. doi:10.1007/978-3-662-47666-6_35.

24 Georg Zetzsche. The complexity of downward closure comparisons. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 123:1–123:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.123.