Unboundedness for Recursion Schemes: A Simpler Type System

David Barozzini
Institute of Informatics, University of Warsaw, Poland

Paweł Parys
Institute of Informatics, University of Warsaw, Poland

Jan Wróblewski
Institute of Informatics, University of Warsaw, Poland

Abstract

Decidability of the problems of unboundedness and simultaneous unboundedness (aka. the diagonal problem) for higher-order recursion schemes was established by Clemente, Parys, Salvati, and Walukiewicz (2016). Then a procedure of optimal complexity was presented by Parys (2017); this procedure used a complicated type system, involving multiple flags and markers. We present here a simpler and much more intuitive type system serving the same purpose. We prove that this type system allows to solve the unboundedness problem for a widely considered subclass of recursion schemes, called safe schemes. For unsafe recursion schemes we only have soundness of the type system: if one can establish a type derivation claiming that a recursion scheme is unbounded then it is indeed unbounded. Completeness of the type system for unsafe recursion schemes is left as an open question. Going further, we discuss an extension of the type system that allows to handle the simultaneous unboundedness problem.

We also design and implement an algorithm that fully automatically checks unboundedness of a given recursion scheme, completing in a short time for a wide variety of inputs.

2012 ACM Subject Classification Theory of computation → Verification by model checking; Theory of computation → Rewrite systems; Mathematics of computing → Lambda calculus

Keywords and phrases Higher-order recursion schemes, boundedness, intersection types, safe schemes

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.112

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version Missing proof details are available in the full version of this paper:
Full Version: `https://arxiv.org/abs/2204.11023`

Supplementary Material Implementation of the presented algorithm is available at:
Software (Source Code): `https://github.com/xi-business/infsat`
archived at `swh:1:dir:075c57b341bf44dc0abeb89fe48ed8aec1691828`

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

1 Introduction

Higher-order recursion schemes (*recursion schemes* for short) proved to be useful in model-checking programs using higher-order functions, see e.g. Kobayashi [13] (recursion schemes are algorithmically manageable abstractions of such programs, faithfully representing the control flow). Higher-order functions are widely used in functional programming languages, like Haskell, OCaml, and Lisp; additionally, higher-order features are now present in most mainstream languages like Java, JavaScript, Python, or C++.

© David Barozzini, Paweł Parys, and Jan Wróblewski;
licensed under Creative Commons License CC-BY 4.0

The formalism of recursion schemes is equivalent via direct translations to simply-typed λY-calculus [27] and to higher-order OI grammars [9, 16]. Collapsible pushdown systems [11] and ordered tree-pushdown systems [6] are other equivalent formalisms. Recursion schemes cover some other models such as indexed grammars [1] and ordered multi-pushdown automata [3].

The usefulness of recursion schemes follows from the fact that trees generated by them have decidable MSO theory [20]. When the property to be verified is given by a parity automaton (a formalism equivalent to the MSO logic), and when the recursion scheme is of order m, the model-checking problem is m-EXPTIME-complete; already for reachability properties the problem is $(m-1)$-EXPTIME-complete [14]. Although this high complexity may be threatening, there exist algorithms that behave well in practice. They make use of appropriate systems of intersection types. Namely, a Japanese group created model-checkers TRecS [13] and HorSat [4], and prototype verification tools, MoCHi [15] and EHMTT verifier [28], on top of them. A group hosted in Oxford created model-checkers HORSC [19] and TravMC2 [18]. Necessarily these model-checkers are very slow on some worst-case examples, but on schemes generated from some real life higher-order programs they usually work in a reasonable time (while in the worst-case a huge type derivation may be required, it is often the case that there exists a small type derivation that can be found quickly).

In recent years, interest has arisen in model checking recursion schemes against properties that are not regular (i.e., not expressible in the MSO logic). This primarily concerns the *unboundedness problem* for word languages recognized by recursion schemes [22], and its generalization – the *simultaneous unboundedness problem* (aka. the diagonal problem) [10, 7, 26]. Decidability of the latter problem implied computability of downward closures (with respect to the subsequence relation) for these languages [29], and decidability of their separability by piecewise testable languages [8]. It was also possible to establish decidability of the WMSO+U logic (an extension of MSO with a quantifier U, talking about unboundedness) over trees generated by recursion schemes [25].

Moreover, there is also a link to asynchronous shared-memory programs, being a common way to manage concurrent requests in a system. In asynchronous programming, each asynchronous function is a sequential program. When run, it can change the global shared state of the program and run other asynchronous functions. A scheduler repeatedly and non-deterministically executes pending asynchronous functions. Majumdar, Thinniyam and Zetzsche [17] have proven that when asynchronous functions are modeled as recursion schemes, the question whether there is an a priori upper bound on the number of pending executions of asynchronous functions can be reduced to the problem we consider here, namely the unboundedness problem for a single recursion scheme.

In this paper we revisit the (simultaneous) unboundedness problem for word languages recognized by recursion schemes. This problem asks, for a set of letters Σ_{a} and a language of words L (recognized by a recursion scheme), whether for every $n \in \mathbb{N}$ there is a word in L where every letter from Σ_{a} occurs at least n times. Equivalently, one can consider a recursion scheme generating an infinite tree t and ask whether for every $n \in \mathbb{N}$ there is a finite branch in t where every letter from Σ_{a} occurs at least n times. The problem is already interesting when $|\Sigma_{\mathsf{a}}| = 1$; then we talk about the unboundedness problem. Decidability of the simultaneous unboundedness problem was first established by Hague, Kochems, and Ong [10], for a well-recognized subclass of recursion schemes, called *safe* schemes [9, 12, 2, 5, 27]. The solution was then generalized to all recursion schemes by Clemente, Parys, Salvati, and Walukiewicz [7]. These two algorithms are useless in practice, not only because their complexity is much higher than the optimal one, but mainly because they perform some transformations of recursion schemes that are extremely costly in every case, not only in the worst case. The complexity of the problem for recursion schemes (namely, $(m-1)$-EXPTIME-completeness for schemes of order m) was settled by Parys [26]; moreover, his solution uses intersection types, and thus it is potentially suitable for implementation (by analogy to the regular model-checking case, where algorithms using type systems led to reasonable implementations). In the single-letter case ($|\Sigma_{\mathsf{a}}| = 1$) the problem is still $(m-1)$-EXPTIME-complete; a slightly simpler type system for this case was presented by Parys [22]. Unfortunately, the type systems of Parys [22, 26] have two drawbacks. First, they are quite complicated: type judgments can be labeled by different kinds of flags and markers, which influence type derivations in a convoluted way. Second (and related to first), it seems that the number of choices for these flags and markers is quite large, and thus finding type derivations even for quite simple recursions schemes may be very costly. Having these drawbacks in mind, we leave experimental evaluation of these algorithms for future work.

In this paper we rather consider a much simpler type system, proposed by Parys in his survey [24], based on an earlier work concerning lambda-terms representing functions on numerals [21]. This type system is much easier than the previous one: every type is labeled only by a single "productivity flag" having an intuitive meaning. Namely, the flag says whether the lambda-term under consideration is responsible for creating occurrences of the letter from Σ_{a}. It was only conjectured that this type system may be used to solve the unboundedness problem.

Our contributions are as follows:

- We prove that the type system allows to solve the unboundedness problem for all safe recursion schemes.
- We show that the algorithm using the type system solves the unboundedness problem for safe recursion schemes of order m in $(m-1)$-EXPTIME (being optimal).
- We prove soundness of the type system for all (i.e., also unsafe) recursion schemes, saying that if one has found a type derivation claiming that a recursion schemes is unbounded then it is indeed unbounded. Completeness of the type system for unsafe recursion schemes is left as an open problem.
- We implement an algorithm solving the unboundedness problem by means of the proposed type system, and we present results of our experiments. The outcome of the algorithm is always correct if the recursion scheme given on input is safe. When the recursion scheme is not safe, proofs of its unboundedness found by the algorithm are still guaranteed to be correct. However, our theorems do not guarantee that the algorithm will find a proof of unboundedness in the unsafe case, so if the algorithm fails to find such a proof, it does not mean that the unsafe recursive scheme is necessarily bounded.
- We then generalize the type system to the simultaneous unboundedness problem (i.e., the multiletter case), obtaining the same properties: algorithm in $(m-1)$-EXPTIME for safe recursion schemes of order m, and soundness for all recursion schemes.
- We implement an optimized version of our algorithm, InfSat, which is fast for a wide variety of inputs. We build upon the implementation of HorSat [4]. We modify HorSat benchmarks to conform to InfSat input format, and present their results to show that InfSat is able to handle inputs described by Broadbent and Kobayashi [4] as "practical" as well as additional benchmarks crafted to measure the speed of InfSat.

2 Preliminaries

The set of *sorts* is constructed from a unique basic sort o using a binary operation $\to$. Thus o is a sort and if α, β are sorts, so is $(\alpha \to \beta)$. The order of a sort is defined by: $\mathit{ord}(\mathsf{o}) = 0$, and $\mathit{ord}(\alpha \to \beta) = \max(1 + \mathit{ord}(\alpha), \mathit{ord}(\beta))$.

A *signature* Σ is a set of typed constants, that is, symbols with associated sorts. We assume that sorts of all constants in the signature are of order at most 1, that is, of the form $\underbrace{\mathsf{o}\to\dots\to\mathsf{o}\to}_{r}\mathsf{o}$; for a constant of such a sort, r is called its *arity*. We fix a distinguished constant $\omega\in\Sigma$ of arity 0 (it will be used in places where a computation diverges).

The set of *infinitary simply-typed lambda-terms* is defined coinductively as follows. A constant $a\in\Sigma$ of sort α is a lambda-term of sort α. For each sort α there is a countable set of variables $x^\alpha, y^\alpha, \dots$ that are also lambda-terms of sort α. If M is a lambda-term of sort β and x^α a variable of sort α then $\lambda x^\alpha.M$ is a lambda-term of sort $\alpha\to\beta$. Finally, if M is a lambda-term of sort $\alpha\to\beta$ and N is a lambda-term of sort α then $M\,N$ is a lambda-term of sort β. As usual, we identify lambda-terms up to alpha-conversion. We often omit the sort annotation of variables, but formally every variable has a sort. We use the standard notions of free variables, substitution, and beta-reduction (of course during substitution and beta-reduction we rename bound variables to avoid name conflicts). A lambda-term is called *closed* when it does not have free variables. For a lambda-term M of sort α, the order of M, denoted $ord(M)$, is defined as $ord(\alpha)$.

The *complexity* of a lambda-term M is the maximum of orders of those subterms of M that are not of the form $a\,M_1\,\dots\,M_k$, where a is a constant and $k\geq 0$ (or 0 if there are no such subterms). Note that for most lambda-terms M the complexity is just the maximum of orders of all subterms of M; the difference is only at complexity 0 and 1: we want lambda-terms built entirely from constants to have complexity 0, not 1.

A closed lambda-term of sort o and complexity 0 is called a *tree*. Equivalently, a lambda-term is a tree if it is of the form $a\,M_1\,\dots\,M_r$, where $M_1,\dots,M_r$ are trees, and r is the arity of a. While talking about a *branch* of a tree, we mean a finite branch that ends in a leaf not being ω-labeled. Formally, a *(finite) branch* of a tree $T=a\,T_1\,\dots\,T_r$ is a sequence of constants $a_1,a_2,\dots,a_k$ such that $a_1=a\neq\omega$, and either $a_2,\dots,a_k$ (with $k\geq 2$) is a finite branch of some T_i, or $r=0$ and $k=1$.

We consider Böhm trees only for closed lambda-terms of sort o. For such a lambda-term M, its *Böhm tree* is constructed by coinduction, as follows: if there is a sequence of beta-reductions from M to a lambda-term of the form $a\,M_1\,\dots\,M_r$, and $T_1,\dots,T_r$ are Böhm trees of $M_1,\dots,M_r$, respectively, then $a\,T_1\,\dots\,T_r$ is a Böhm tree of M; if there is no such a sequence of beta-reductions from M, then the constant ω is a Böhm tree of M. It is folklore that every closed lambda-term of sort o has exactly one Böhm tree (the order in which beta-reductions are performed does not matter); this tree is denoted by $BT(M)$. Notice that a Böhm tree is indeed a tree, and that if M is finite then its Böhm tree equals the beta-normal form of M.

A *recursion scheme* $\mathcal{G}$ is a finite representation of a closed lambda-term $\Lambda(\mathcal{G})$ that is of sort o and regular, that is, has finitely many different subterms. We postpone the definition of a recursion scheme until Section 6. As the *order* of $\mathcal{G}$ we understand the complexity of $\Lambda(\mathcal{G})$. We do not claim that every regular lambda-term of sort o can be directly represented by a recursion scheme, however we remark that every regular lambda-term of sort o is equivalent (in the sense of having the same Böhm tree) to some lambda-term represented by a recursion scheme.

Simultaneous unboundedness problem. Fix a set of *important constants* Σ_{a}. We say that a closed lambda-term M of sort o is *unbounded* (with respect to Σ_{a}) if for every $n\in\mathbb{N}$ there exists a finite branch of $BT(M)$ with at least n occurrences of every constant from Σ_{a}. The *simultaneous unboundedness problem* (SUP) is to decide, given a recursion scheme $\mathcal{G}$, whether $\Lambda(\mathcal{G})$ is unbounded.

In the special case of $|\Sigma_{\mathsf{a}}|=1$ we talk about the *unboundedness problem.*

3 Type system

In this section we present a type system that allows us to solve the (single-letter) unboundedness problem. The type system was first proposed in Parys' survey [24], without any correctness proofs.

In the remaining part of the paper, except for Section 7, we assume that the signature Σ consists of four constants: a of arity 1, b of arity 2, and c and ω of arity 0, where only a is important, that is $\Sigma_{\mathsf{a}} = \{\mathsf{a}\}$. This is without loss of generality, since we can replace a constant of arbitrary arity by a combination of these four constants, without changing the answer to the unboundedness problem.

Before defining the type system (and safety), let us state a theorem describing its desired properties:

▶ **Theorem 3.1.** *The following two statements are equivalent for every safe closed lambda-term M of sort o:*
(1) *M is unbounded (i.e., for every $n \in \mathbb{N}$ there exists a finite branch of $BT(M)$ with at least n occurrences of the constant a);*
(2) *for every $n \in \mathbb{N}$ there exists $v \geq n$ such that one can derive $\emptyset \vdash M : (v, \mathsf{r})$.*

In this theorem, type judgments contain a natural number v, called a *productivity value*. The goal of this value is to approximately count the number of occurrences of the important constant a on a selected branch of $BT(M)$.

Types in the type system differ from sorts in that on the left side of $\rightarrow$, instead of a single type, we have a set of so-called *type pairs* (f, τ), where τ is a *type*, and $f \in \{\mathsf{pr}, \mathsf{np}\}$ is a *productivity flag* (where pr stands for productive, and np for nonproductive). The unique atomic type is denoted r. More precisely, for each sort α we define the set $\mathcal{T}^\alpha$ of types of sort α as follows:

$$\mathcal{T}^{\mathsf{o}} = \{\mathsf{r}\}, \qquad\qquad \mathcal{T}^{\alpha\rightarrow\beta} = \mathcal{P}(\{\mathsf{pr}, \mathsf{np}\} \times \mathcal{T}^\alpha) \times \mathcal{T}^\beta,$$

where $\mathcal{P}$ denotes the powerset. A type $(T, \tau) \in \mathcal{T}^{\alpha\rightarrow\beta}$ is denoted as $\bigwedge T \rightarrow \tau$, or $\bigwedge_{i\in I}(f_i, \tau_i) \rightarrow \tau$ when $T = \{(f_i, \tau_i) \mid i \in I\}$. In this notation we implicitly assume that all the pairs (f_i, τ_i) are different. The empty intersection is denoted by $\top$. Moreover, to our terms we will not only assign a type τ, but also a productivity flag $f \in \{\mathsf{pr}, \mathsf{np}\}$ (which together form a pair (f, τ)). Let us emphasize that for every sort α the set $\mathcal{T}^\alpha$ is finite.

Intuitively, a lambda-term has type $\bigwedge T \rightarrow \tau$ when it can return τ, while taking an argument for which we can derive all type pairs from T; simultaneously, while having such a type, the lambda-term is obligated to use its arguments in all ways described by type pairs from T. For example, the lambda-term $\lambda x.\mathsf{c}$ does not use its argument, and hence it is necessarily of type $\top \rightarrow \mathsf{r}$ (i.e., $\bigwedge T \rightarrow \tau$ with $T = \emptyset$).

To determine the productivity flag f assigned to a lambda-term M, we should imagine that M is a subterm of a closed term K of sort o, and we should select some finite branch in $BT(K)$ (with the intuition that different choices of the branch correspond to different type derivations). Then, we assign to M the flag pr (productive) when the subterm M is responsible for increasing the number of occurrences of the constant a on the selected branch. To be more precise, a lambda-term is responsible for producing occurrences of a constant a in two cases. First, when it explicitly contains the constant a – assuming that this a will be placed on the selected branch. Second, when it takes a productive argument (i.e., an argument responsible for producing a) and uses it at least twice. The first possibility occurs for example in the lambda-term $M_1 = \lambda x.\mathsf{a}\,x$; the constant a is explicitly produced. In order to see the second possibility, consider the lambda-term $M_2 = \lambda y.\lambda x.y\,(y\,x)$, and suppose that the argument received for y is the aforementioned productive lambda-term $\lambda x.\mathsf{a}\,x$, which outputs the constant a. Then, the lambda-term M_2 is itself responsible for increasing the number of occurrences of a in the resulting tree, compared to the number of occurrences produced by the argument. Notice that the same lambda-term M_2 has also another type: the argument received for y may be nonproductive (say $\lambda x.x$), and then M_2 becomes nonproductive as well. Next, let us compare M_2 with the lambda-term $M_2' = \lambda y.\lambda x.y\,x$, when used with the argument $\lambda x.\mathsf{a}\,x$. Although the argument is used, and one a is output, the lambda-term M_2' has nothing to do with increasing the number of occurrences of a; it is nonproductive.

A *type judgment* is of the form $\Gamma \vdash M : (v, \tau)$, where we require that the type τ and the lambda-term M are of the same sort. The *type environment* Γ is a set of bindings of variables of the form $x^\alpha : (g, \tau)$, where $g \in \{\mathsf{pr}, \mathsf{np}\}$ and $\tau \in \mathcal{T}^\alpha$. In Γ we may have multiple bindings for the same variable. By $\mathit{dom}(\Gamma)$ we denote the set of variables x which are bound by Γ, and by $\Gamma{\restriction}_{\mathsf{pr}}$ we denote the set of only those bindings $x : (g, \tau)$ from Γ in which $g = \mathsf{pr}$. We are not only interested in whether some type can be derived, but we also want to assign a value to every derivation; to this end, a type judgment contains a number $v \in \mathbb{N}$, which is called a *productivity value.* Having $v = 0$ corresponds to the np flag, while positive values correspond to the pr flag. Thus, while a productivity flag says only whether any occurrence of the important constant a is produced, the productivity value approximates (is a lower bound on) the number of produced occurrences of this constant. Note that in type judgments we store the productivity value, coming from the infinite set $\mathbb{N}$, while in types we abstract this value to the productivity flag, which allows us to have finitely many types.

We now present rules of the type system, starting from rules for constants:

$$\emptyset \vdash \mathsf{a} : (1, (f, \mathsf{r}) \to \mathsf{r}) \qquad\qquad \emptyset \vdash \mathsf{c} : (0, \mathsf{r})$$

$$\emptyset \vdash \mathsf{b} : (0, (f, \mathsf{r}) \to \top \to \mathsf{r}) \qquad\qquad \emptyset \vdash \mathsf{b} : (0, \top \to (f, \mathsf{r}) \to \mathsf{r})$$

Notice that in the rule for b we have a type (f, r) (with an arbitrary flag f) only for one of the two arguments; this corresponds to the fact that we are interested in a single branch of the Böhm tree, so we want to descend only to a single child. Moreover, we do not have a rule for the constant ω, because by definition a branch cannot contain occurrences of this constant.

While typing a variable x, we take its type from the type environment, and we use 0 as the productivity value. The lambda-term x itself is not responsible for producing any constants, no matter whether the lambda-term substituted for x will produce any constants or not. A lambda-term becomes productive when a productive variable x is used twice; we account for that in the application typing rule (@).

$$x : (f, \tau) \vdash x : (0, \tau)$$

When we pass through a lambda-binder, we simply move some type pairs between the argument and the type environment:

$$\frac{\Gamma \cup \{x : (f_i, \tau_i) \mid i \in I\} \vdash K : (v, \tau) \qquad x \notin \mathit{dom}(\Gamma)}{\Gamma \vdash \lambda x.K : (v, \bigwedge_{i\in I}(f_i, \tau_i) \to \tau)}\ (\lambda)$$

Before giving the last rule, we need one more definition. Given a family of type environments $(\Gamma_i)_{i\in J}$, a *duplication factor*, denoted $\mathit{dupl}((\Gamma_i)_{i\in J})$, equals $\sum_{i\in J} \left|\Gamma_i{\restriction}_{\mathsf{pr}}\right| - \left|\bigcup_{i\in J} \Gamma_i{\restriction}_{\mathsf{pr}}\right|$. It counts the number of repetitions ("duplications") of productive type bindings in the type environments: a productive type binding belonging to one type environment does not add anything, a productive type binding belonging to two type environments adds 1, and so on.

$$\frac{0 \notin I \qquad \forall i \in I.\ (f_i = \mathsf{pr}) \Leftrightarrow (v_i > 0 \lor \Gamma_i{\restriction}_{\mathsf{pr}} \neq \emptyset) \qquad \Gamma_0 \vdash K : (v_0, \bigwedge_{i\in I}(f_i,\tau_i) \to \tau) \qquad \Gamma_i \vdash L : (v_i,\tau_i) \text{ for each } i \in I}{\bigcup_{i\in\{0\}\cup I} \Gamma_i \vdash K\,L : (\mathit{dupl}((\Gamma_i)_{i\in\{0\}\cup I}) + \sum_{i\in\{0\}\cup I} v_i, \tau)}\ (@)$$

Here, by using the notation $\bigwedge_{i\in I}(f_i,\tau_i)$, we assume that the pairs (f_i,τ_i) are all different.

In the rule above, the condition $(f_i = \mathsf{pr}) \Leftrightarrow (v_i > 0 \lor \Gamma_i{\restriction}_{\mathsf{pr}} \neq \emptyset)$ means that when K requires a "productive" argument, then either we can apply an argument L that is itself productive, or we can apply a nonproductive L that uses a productive variable (the argument obtained after substituting something for this variable will become productive).

Using the *dupl* function we realize the intuition that when a variable responsible for creating occurrences of a (i.e., productive) is used at least twice, then the lambda-term is itself responsible for increasing the number of occurrences of a; thus we add *dupl* to the productivity value.

Because we are interested in counting duplications of type bindings, it was necessary to require that every type binding from the type environment is actually used somewhere (in particular $\Gamma \vdash M : (f,\tau)$ does not necessarily imply $\Gamma, x : (g,\sigma) \vdash M : (f,\tau)$). On the other hand, a type environment is a set: a repeated usage of a type binding is counted in the productivity value, but is not reflected in the type environment.

Let us underline that although we consider infinite lambda-terms, all type derivations are required to be finite. This may look suspicious, but note that when a lambda-term has type $\top \to \tau$ (like, e.g., the constant b), then we need no derivation for its argument.

▶ **Example 3.2.** Let us give an example of a derivation for a lambda-term $\lambda y.\lambda z.y\,(y\,(\mathsf{a}\,z))$ of sort $(\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$. In this particular derivation, we will assume that y and z are both productive.

$$\frac{\vdash \mathsf{a} : (1, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \qquad z : (\mathsf{pr},\mathsf{r}) \vdash z : (0,\mathsf{r})}{z : (\mathsf{pr},\mathsf{r}) \vdash \mathsf{a}\,z : (1,\mathsf{r})}\ (@)$$

In the innermost application, we have an important constant a and a productive variable z. The important constant has value 1. The value of z is 0 because, even though it is productive, any important constants it produces will be just moved from the argument substituted for z. No important constant will be lost or produced during this process. There are no duplicates of variables in the application $\mathsf{a}\,z$, so the value of the application is equal to sum of values, that is, 1.

$$\frac{y : (\mathsf{pr}, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash y : (0, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \qquad z : (\mathsf{pr},\mathsf{r}) \vdash \mathsf{a}\,z : (1,\mathsf{r})}{z : (\mathsf{pr},\mathsf{r}),\ y : (\mathsf{pr}, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash y\,(\mathsf{a}\,z) : (1,\mathsf{r})}\ (@)$$

We apply y to $\mathsf{a}\,z$. Variable y is productive and takes a productive argument, which means that it incorporates its argument into the tree it produces and somehow increases the number of important constants in the process. However, this increase is computed in the lambda-term that is substituted for y, not here, hence it has value 0.

$$\frac{y : (\mathsf{pr}, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash y : (0, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \qquad z : (\mathsf{pr},\mathsf{r}),\ y : (\mathsf{pr}, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash y\,(\mathsf{a}\,z) : (1,\mathsf{r})}{z : (\mathsf{pr},\mathsf{r}),\ y : (\mathsf{pr}, (\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash y\,(y\,(\mathsf{a}\,z)) : (2,\mathsf{r})}\ (@)$$

Both sides have y in the environment here, so the duplication factor in $y\,(y\,(\mathsf{a}\,z))$ is 1. We add it to the sum of values from both sides of the application, obtaining 2.

$$\dfrac{\dfrac{z : (\mathsf{pr},\mathsf{r}),\; y : (\mathsf{pr},(\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash y\,(y\,(\mathsf{a}\,z)) : (2,\mathsf{r})}{y : (\mathsf{pr},(\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \vdash \lambda z.y\,(y\,(\mathsf{a}\,z)) : (2,(\mathsf{pr},\mathsf{r}) \to \mathsf{r})}\,(\lambda)}{\vdash \lambda y.\lambda z.y\,(y\,(\mathsf{a}\,z)) : (2,(\mathsf{pr},(\mathsf{pr},\mathsf{r}) \to \mathsf{r}) \to (\mathsf{pr},\mathsf{r}) \to \mathsf{r})}\,(\lambda)$$

The lambda-binders move z and y from the environment into the type without changing the value. The final value for this closed lambda-term is 2, as it always adds one important constant to the Böhm tree and, when something that increases the number of important constants is substituted for y, it applies that twice, causing at least one extra important constant to be added compared to what just one instance of y would do.

Note that it is possible to derive other type judgments for the lambda-term $y\,(y\,(\mathsf{a}\,z))$. For example, the value would be equal to one if y was not productive, even if z was productive:

$$y : (\mathsf{np},(\mathsf{pr},\mathsf{r}) \to \mathsf{r}),\; z : (\mathsf{pr},\mathsf{r}) \vdash y\,(y\,(\mathsf{a}\,z)) : (1,\mathsf{r}).$$

The rationale is that while y takes productive arguments, it uses them exactly once and does not add an important constant to the Böhm tree either, so applying it any number of times will not change the number of important constants.

4 Soundness

In this section we prove the soundness of the type system, as described by following lemma, from which the (2) ⇒ (1) implication of Theorem 3.1 follows immediately:

▶ **Lemma 4.1.** *If we can derive $\emptyset \vdash M : (v,\mathsf{r})$, where M is a closed lambda-term of sort o, then $BT(M)$ has a finite branch containing at least v occurrences of a.*

We prove Lemma 4.1 as follows. First, as in work of Parys [26, 24] we observe that instead of working directly with infinite lambda-terms M, we can "cut off" parts of M not involved in the finite derivation of $\emptyset \vdash M : (v,\mathsf{r})$, (i.e., subterms used as arguments for which no type pair is required). Formally, by cutting off we mean replacing by lambda-terms of the form $\lambda x_1.\cdots.\lambda x_k.\omega$, where the variables $x_1,\dots,x_k$ are chosen so that the sort of the lambda-term is appropriate. In consequence, it is enough to prove Lemma 4.1 for finite lambda-terms.

Second, we repeatedly use Lemma 4.2 below, reducing M to its beta-normal form N, and never decreasing the productivity value. Finally, we observe that the beta-normal form N is simply a tree; a derivation of $\emptyset \vdash N : (v',\mathsf{r})$, using only the rules for constants and application, describes some branch of this tree containing exactly v' occurrences of the important constant a. It remains to justify the following lemma, which describes a single beta-reduction:

▶ **Lemma 4.2.** *If we can derive $\emptyset \vdash M : (v,\mathsf{r})$, where M is a finite closed lambda-term of sort o, and M is not in the beta-normal form, then we can derive $\emptyset \vdash N : (v',\mathsf{r})$ for a lambda-term N such that $M \to_\beta N$, and for some v' satisfying $v \le v'$.*

While proving this lemma, we consider the leftmost outermost redex, $(\lambda x.K)\,L$. Thanks to this choice, L is necessarily a closed subterm. This simplifies the situation: it is enough to consider empty type environments (we remark, however, that Lemma 4.2 can be shown in a similar way for every reduction, not only for the leftmost outermost reduction). We want to replace every subderivation D for a type judgment $\emptyset \vdash (\lambda x.K)\,L : (w,\tau)$ concerning this redex with a derivation D' for $\Gamma \vdash K[L/x] : (w', \tau)$. We obtain D' by appropriately reorganizing subderivations of D: we take the subderivation of D concerning K, we replace every leaf deriving a type σ for x by the subderivation of D deriving this type σ for L, and we update type environments and productivity values appropriately.

Notice that every subderivation concerning L is moved to at least one leaf concerning x (nothing can disappear). The only reason why the value of the derivation can decrease is that potentially a productive type binding $x : (\mathsf{pr}, \sigma)$ was duplicated (say, n times) in the derivation concerning K. In D' this binding is no longer present (in $K[L/x]$ there is no x) so the value decreases by n, but in this situation the subderivation deriving σ for L becomes inserted in $n + 1$ leaves. This subderivation is productive, so by creating n additional copies of this subderivation we increase the value at least by n, compensating the loss caused by elimination of x. This implies that $w \leq w'$, and consequently $v \leq v'$. Check Appendix A of the full version for more details.

5 Completeness for safe lambda-terms

In this section we prove completeness for a subclass of lambda-terms called safe lambda-terms. The question whether completeness holds for general lambda-terms is left open.

A lambda-term M is *superficially safe* when for every free variable x of M it holds that $\mathit{ord}(M) \leq \mathit{ord}(x)$. A lambda-term M is *safe* if it is superficially safe, and if for its every subterm of the form $K\, L_1\, \ldots\, L_k$, where K is not an application and $k \geq 1$, all subterms $K, L_1, \ldots, L_k$ are superficially safe. This definition of safety coincides with the definitions from Salvati and Walukiewicz [27], and from Blum and Ong [2].

Completeness for safe lambda-terms is given by the following lemma.

▶ **Lemma 5.1.** *For every $m \in \mathbb{N}$ there exists a function $H_m \colon \mathbb{N} \to \mathbb{N}$ such that if M is a closed safe lambda-term of sort o and complexity at most m, and in $BT(M)$ there is a finite branch having at least n occurrences of a, then we can derive $\emptyset \vdash M : (v, \mathsf{r})$ for some v such that $n \leq H_m(v)$.*

While proving this lemma, it is convenient to split the productivity value into two parts. Given some fixed $\ell \in \mathbb{N}$, instead of type judgments of the form $\Gamma \vdash N : (v, \tau)$ we consider extended type judgments of the form $\Gamma \vdash N : (u \oplus_\ell w, \tau)$, where $u + w = v$. On u we accumulate only duplication factors concerning variables of order at least ℓ, and on w the remaining part of the value (i.e., duplication factors concerning variables of order smaller than ℓ, plus the number of rules for the constant a). Having this definition, we can state a counterpart of Lemma 4.2:

▶ **Lemma 5.2.** *Suppose that we can derive $\Gamma \vdash K[L/x] : (u \oplus_\ell w, \tau)$, where L is closed, has order ℓ, and does not use any variables of order at least ℓ. Then we can derive $\Gamma \vdash (\lambda x.K)\, L : (u' \oplus_\ell w', \tau)$ for some u', w' such that $2^u \cdot w \leq 2^{u'} \cdot w'$ and $u + w = 0 \Rightarrow u' + w' = 0$.*

Similarly to Lemma 4.2, the derivation concerning $(\lambda x.K)\, L$ in the above lemma is obtained by appropriately reorganizing subderivations of the derivation concerning $K[L/x]$. In the type derivation concerning $K[L/x]$, there are some (zero or more) subderivations concerning L. A difficulty is caused by the fact that the same type pair (f, σ) may be derived for multiple copies of L in $K[L/x]$, using different subderivations. For every such (f, σ) derived for L we should choose just one subderivation, so that we can use it for the only copy of L in $(\lambda x.K)\, L$. We choose the subderivation that provides the largest second component of the value; the other subderivations are removed. The value of the new derivation decreases, because some subderivations are removed, and increases, because we have a new variable x, that may cause some duplications.

It remains to see that the value $u' \oplus_\ell w'$ of the new derivation satisfies the inequality $2^u \cdot w \leq 2^{u'} \cdot w'$. Consider some type pair (f, σ) derived for L. If $f = \mathsf{np}$, the value of the removed subderivations is 0, and the duplications of $x : (f, \sigma)$ in the type environment are not counted, so such a type pair does not cause any change of the value. Suppose that $f = \mathsf{pr}$, and that we have removed n subderivations proving the type pair (f, σ) for L; this may decrease the second component of the value at most $n + 1$ times (the n removed subderivations have values not greater than the one that remains). Simultaneously, in K we use $n + 1$ times the variable x with this type pair (f, σ), which increases the value (the total duplication factor) by n. The key point is that L does not use any variables of order at least ℓ, and thus the first component (concerning variables of order at least ℓ) of the value of subderivations for L is 0. Thus, we decrease the second component of the value at most $n + 1$ times, and we increase the first component of the value by at least n. Since $2^n \geq n + 1$, we obtain the required inequality between values.

It is also important that L is closed so that after removing some subderivations concerning L, the type environment of the whole derivation remains unchanged. This finishes the proof of Lemma 5.2 (a more formal proof, containing all details, can be found in Appendix B.1 of the full version).

We now come back to the proof of Lemma 5.1. First, as in the previous section, we can assume that M is finite by "cutting off" (i.e., replacing with $\lambda x_1.\cdots.\lambda x_k.\omega$) its parts not needed for producing the selected finite branch of $BT(M)$.

Second, it is convenient to assume here that M is homogeneous. A sort $\alpha_1 \to \cdots \to \alpha_k \to \mathsf{o}$ is *homogeneous* if $\mathit{ord}(\alpha_1) \geq \cdots \geq \mathit{ord}(\alpha_k)$ and all $\alpha_1, \ldots, \alpha_k$ are homogeneous. A lambda-term is homogeneous if its every subterm has a homogeneous sort. It is known that every finite closed safe lambda-term M of sort o can be converted into a lambda-term M' that is additionally homogeneous [23], but has the same beta-normal form. Analyzing how M is transformed into M' (in [23]), it is tedious but straightforward to check that we can derive $\emptyset \vdash M : (v, \mathsf{r})$ if and only if we can derive $\emptyset \vdash M' : (v, \mathsf{r})$; this is done in Appendix B.2 of the full version.

It is thus enough to prove Lemma 5.1 assuming that M is finite and homogeneous. To this end, we consider a particular sequence of reductions leading from M to its beta-normal form $BT(M)$. Namely, whenever a lambda-term N reached so far (i.e., after some number of reductions) from M is of complexity $\ell + 1$, then we reduce in N a redex $(\lambda x.K)\, L$ such that $\mathit{ord}(\lambda x.K) = \ell + 1$, and no variables of order at least ℓ occur in L, both as free variables and as bound variables; we call such a redex *ℓ-good*. Such a redex always exists: among redexes with $\mathit{ord}(\lambda x.K) = \ell + 1$ it is enough to choose the rightmost one. The complexity of a lambda-term cannot increase during a beta-reduction, thus in the sequence of reductions we can find lambda-terms $M_m, M_{m-1}, \ldots, M_0$, where $M_m = M$, and $M_0 = BT(M)$, and the complexity of every $M_{\ell+1}$ is at most $\ell + 1$, and M_ℓ can be reached from $M_{\ell+1}$ by a sequence of ℓ-good reductions.

Homogeneity is preserved during beta-reductions. Moreover, ℓ-good beta-reductions preserve safety (while this is not true for arbitrary beta-reductions). Indeed, homogeneity for an ℓ-good redex $(\lambda x.K)\, L$ implies that $\mathit{ord}(L) = \ell$ (the first argument is of the highest order, i.e., of order ℓ), thus safety implies that all free variables of L are of order at least ℓ. But, by the definition of an ℓ-good redex, no variables of order at least ℓ occur in L. It follows that L is closed. While substituting a closed lambda-term L for x, all superficially safe subterms of K remain superficially safe (no new free variables appear). Thus the lambda-term after the ℓ-good beta-reduction remains safe.

Recall the sequence of lambda-terms $M_m, M_{m-1}, \dots, M_0$ defined above. Assuming that in $BT(M) = M_0$ there is a finite branch having exactly n occurrences of a, we can derive $\emptyset \vdash M_0 : (n, \mathsf{r})$ using only the rules for application and constants. Suppose now that we can derive $\emptyset \vdash M_\ell : (v_\ell, \mathsf{r})$ for some $\ell \in \{0, \dots, m-1\}$. We want to find a derivation of $\emptyset \vdash M_{\ell+1} : (v_{\ell+1}, \mathsf{r})$, where for arbitrarily large values v_ℓ, the values $v_{\ell+1}$ are also arbitrarily large (more precisely, we obtain the inequality $v_\ell \le 2^{v_{\ell+1}}$). Notice that between $M_{\ell+1}$ and M_ℓ there may be arbitrarily many (ℓ-good) reductions. Consider thus some ℓ-good redex $(\lambda x.K)\, L$ that is reduced at some moment between $M_{\ell+1}$ and M_ℓ. By definition, L does not use any variables of order at least ℓ and, as already observed, L is closed; thus, Lemma 5.2 can be applied. We start with $\emptyset \vdash M_\ell : (0 \oplus_\ell v_\ell, \mathsf{r})$ (notice that in M_ℓ there are no variables of order ℓ or higher, so the first component of the value is 0). Then, we use Lemma 5.2 for every ℓ-good beta-reduction between $M_{\ell+1}$ and M_ℓ. This leads to $\emptyset \vdash M_{\ell+1} : (u_{\ell+1} \oplus_\ell w_{\ell+1}, \mathsf{r})$, where $v_\ell \le 2^{u_{\ell+1}} \cdot w_{\ell+1} \le 2^{v_{\ell+1}}$ (taking $v_{\ell+1} = u_{\ell+1} + w_{\ell+1}$), as required.

The function H_m, appearing in the statement of Lemma 5.1, can be defined as a tower of powers of 2 of height m: $H_0(v) = v$ and $H_{\ell+1}(v) = H_\ell(2^v)$. Then from $n \le H_\ell(v_\ell)$ it follows that $n \le H_{\ell+1}(v_{\ell+1})$; since $n = H_0(v_0)$, we thus obtain the desired inequality $n \le H_m(v_m)$.

6 The algorithm

Both design and implementation of our algorithm is based on the HorSat algorithm, a saturation-based algorithm for model checking recursion schemes against alternating automata by Broadbent and Kobayashi [4].

Our type system was described in previous sections to work with any infinitary lambda-term. The algorithm, in turn, inputs infinitary lambda-terms represented in a finite way, in the form of a recursion scheme. To be concrete, we make this representation explicit now: A recursion scheme $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, X_{\mathsf{st}})$ consists of

- a signature Σ (i.e., a set of constants with assigned arities),
- a set $\mathcal{N}$ of nonterminals with assigned sorts (formally, nonterminals are just distinguished variables),
- a mapping $\mathcal{R}$ from nonterminals in $\mathcal{N}$ to finite lambda-terms such that $\mathcal{R}(X)$ is of the same sort as X, has no free variables other than nonterminals from $\mathcal{N}$, and is of the form $\lambda x_1.\cdots.\lambda x_n.K$, where the subterm K does not contain any lambda-binders, and
- a starting nonterminal $X_{\mathsf{st}} \in \mathcal{N}$ of sort o.

We assume that elements of $\mathcal{N}$ are not used as bound variables, and that $\mathcal{R}(X)$ is not a nonterminal. Furthermore, as in previous sections, we assume for simplicity that $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, where the constants $\mathsf{a}, \mathsf{b}, \mathsf{c}$ have arity $1, 2, 0$, respectively (the implementation, however, accepts arbitrary signatures).

Given a recursion scheme $\mathcal{G}$, and a lambda-term M (possibly containing some nonterminals from $\mathcal{N}$), let $\Lambda_{\mathcal{G}}(M)$ be the lambda-term obtained as a limit of applying repeatedly the following operation to M: take an occurrence of some nonterminal X, and replace it by $\mathcal{R}(X)$ (the nonterminals should be chosen so that every nonterminal is eventually replaced). We remark that while substituting $\mathcal{R}(X)$ for a nonterminal X, there is no need for any renaming of variables (capture-avoiding substitution), since $\mathcal{R}(X)$ does not have free variables other than nonterminals. The infinitary lambda-term *represented by* $\mathcal{G}$ is defined as $\Lambda_{\mathcal{G}}(X_{\mathsf{st}})$, and denoted $\Lambda(\mathcal{G})$. Observe that $\Lambda(\mathcal{G})$ is a closed lambda-term of sort o. We say that $\mathcal{G}$ is *safe* if $\mathcal{R}(X)$ is safe for every $X \in \mathcal{N}$; then $\Lambda(\mathcal{G})$ is safe as well.

The goal of the algorithm is to determine whether $\Lambda(\mathcal{G})$ for a given recursion scheme $\mathcal{G}$ is unbounded or not. In the light of Theorem 3.1, this boils down to checking whether one can derive $\emptyset \vdash \Lambda(\mathcal{G}) : (v, \mathsf{r})$ for arbitrarily large values of v. The idea is to find a single derivation with a productive "loop"; by repeating this loop, one can increase the productivity value arbitrarily. We make this more precise now.

First, we determine type pairs that can be derived for non-terminals. Namely, for $v \in \mathbb{N}$ let $\mathfrak{p}(v) = \mathsf{pr}$ if $v > 0$ and $\mathsf{pr}(v) = \mathsf{np}$ otherwise. Let $\mathcal{T}_\mathcal{G}$ be the smallest set containing all bindings $X : (\mathfrak{p}(v), \tau)$ (with $X \in \mathcal{G}$) such that we can derive $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$, potentially using $\emptyset \vdash Y : (w, \sigma)$ for $(Y : (\mathfrak{p}(w), \sigma)) \in \mathcal{T}_\mathcal{G}$ as assumptions (i.e., as additional typing rules).

Note that productivity values in derivations may be shifted (i.e., increased/decreased by a constant): we can derive $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$ out of an assumption $\emptyset \vdash Y : (w, \sigma)$ if and only if we can derive $\emptyset \vdash \mathcal{R}(X) : (v + (w' - w), \tau)$ out of an assumption $\emptyset \vdash Y : (w', \sigma)$ with $\mathfrak{p}(w) = \mathfrak{p}(w')$. It is thus enough to try assumptions $\emptyset \vdash Y : (w, \sigma)$ with $w = 0$ and $w = 1$ only.

We have only finitely many bindings that may be potentially added to $\mathcal{T}_\mathcal{G}$, as there are finitely many possible types per sort and each nonterminal of $\mathcal{G}$ has a fixed sort. Moreover, taking into account the above, there are only finitely many derivations that may be potentially created for lambda-terms $\mathcal{R}(X)$. It follows that $\mathcal{T}_\mathcal{G}$ may be computed using saturation (i.e., as the least fixed point). The following lemma is immediate:

▶ **Lemma 6.1.** *For any nonterminal $X \in \mathcal{N}$ and any type pair (f, τ) one can derive $\Lambda_\mathcal{G}(X) : (v, \tau)$ for some v with $\mathfrak{p}(v) = f$ if and only if $(X : (f, \tau)) \in \mathcal{T}_\mathcal{G}$.*

Next, knowing which type judgments may be derived, we want to detect a productive cycle. To this end, we create a graph with bindings from $\mathcal{T}_\mathcal{G}$ as nodes, called a *derivation graph*. We draw an edge from $X : (\mathfrak{p}(v), \tau)$ to $Y : (\mathfrak{p}(w), \sigma)$ if one can derive $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$ using $\emptyset \vdash Y : (w, \sigma)$ as an assumption, and potentially using some other assumptions $\emptyset \vdash Z : (u, \rho)$ with $(Z : (\mathfrak{p}(u), \rho)) \in \mathcal{T}_\mathcal{G}$ (the assumption $\emptyset \vdash Y : (w, \sigma)$ necessarily has to be used, at least once). Such an edge is called *productive* if $v > w$. Note that this may happen for three reasons: 1) in the derivation there is some positive duplication factor or an important constant, 2) some other assumption $\emptyset \vdash Z : (u, \rho)$ with $u > 0$ is used, or 3) the assumption $\emptyset \vdash Y : (w, \sigma)$ is used more than once and $w > 0$. Note that edges of the derivation graphs can be incrementally computed at the time of computing $\mathcal{T}_\mathcal{G}$. We now obtain the main theorem, adding Point (3) equivalent to Point (2) from Theorem 3.1:

▶ **Theorem 6.2.** *The following two statements are equivalent for every recursion scheme $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, X_{\mathsf{st}})$:*

(2) *one can derive $\emptyset \vdash \Lambda(\mathcal{G}) : (v, \mathsf{r})$ for arbitrarily large values of v,*

(3) *$(X_{\mathsf{st}} : (\mathsf{pr}, \mathsf{r})) \in \mathcal{T}_\mathcal{G}$ and the derivation graph contains a cycle with a productive edge, reachable from $X_{\mathsf{st}} : (\mathsf{pr}, \mathsf{r})$.*

By Theorem 3.1, assuming additionally that $\mathcal{G}$ is safe, Point (2) (and hence Point (3) as well) is also equivalent to the property we want to check:

(1) $\Lambda(\mathcal{G})$ is unbounded.

Proof. Point (2) follows from Point (3) quite directly. Indeed, an edge from $X : (\mathsf{pr}, \tau)$ to $Y : (\mathsf{pr}, \sigma)$ means that any derivation of $\emptyset \vdash \Lambda_\mathcal{G}(Y) : (w, \sigma)$ can be extended to a derivation of $\emptyset \vdash \Lambda_\mathcal{G}(X) : (v, \tau)$ for some $v \geq w$ (in the sense that the latter contains the former as a subderivation), where $v > w$ if the edge is productive. Following edges on the cycle with a productive edge we can thus extend a derivation of $\emptyset \vdash \Lambda_{\mathcal{G}}(X) : (v, \tau)$ into a larger derivation of $\emptyset \vdash \Lambda_{\mathcal{G}}(X) : (v', \tau)$, where $v' > v$; doing this repeatedly increases the productivity value arbitrarily. Finally, we can use the path from $X_{\mathsf{st}} : (\mathsf{pr}, \mathsf{r})$ to the cycle, making the created derivation a part of a derivation concerning $\Lambda(\mathcal{G}) = \Lambda_{\mathcal{G}}(X_{\mathsf{st}})$.

Let us move on to the implication from Point (2) to Point (3). First, consider all possible derivations of $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$ (with arbitrary $X \in \mathcal{N}$, v, τ) that may use assumptions (additional typing rules) $\emptyset \vdash Y : (v', \tau')$ for nonterminals Y, as in the definition of $\mathcal{T}_{\mathcal{G}}$. Because $\mathcal{R}(X)$ for every $X \in \mathcal{N}$ is a finite lambda-term, and $|\mathcal{N}|$ is finite as well, there exists a bound B such that any derivation as above uses assumptions for nonterminals in at most B leaves, and simultaneously the growth of productivity value caused by duplication factor or important constants in the derivation is at most B.

Next, we prove (by induction on the size of a derivation) that if we can derive $\emptyset \vdash \Lambda_{\mathcal{G}}(X) : (v, \tau)$ with $v \geq (2B)^k$ for some $k \in \mathbb{N}$, then $(X : (\mathfrak{p}(v), \tau)) \in \mathcal{T}_{\mathcal{G}}$ and in the derivation graph there is a path from this node having at least k productive edges. Indeed, out of a derivation of $\emptyset \vdash \Lambda_{\mathcal{G}}(X) : (v, \tau)$ we can reconstruct a derivation of $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$; whenever the former derivation contains a (strictly smaller) subderivation concerning $\Lambda_{\mathcal{G}}(Y)$ for some nonterminal Y, in the latter we use an assumption concerning Y, which is in $\mathcal{T}_{\mathcal{G}}$ by the induction hypothesis. This already shows that $(X : (\mathfrak{p}(v), \tau)) \in \mathcal{T}_{\mathcal{G}}$. If $k = 0$, we are done. Suppose that $k \geq 1$. By properties of the bound B, in the derivation of $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$ there has to be an assumption $\emptyset \vdash Y : (v', \tau')$ with $v' \geq \frac{v-B}{B} \geq \frac{v}{2B}$ (from v we subtract B for duplication factors and important constants inside the derivation of $\emptyset \vdash \mathcal{R}(X) : (v, \tau)$, and we divide the remaining part of the productivity value into at most B assumptions), that is, we can derive $\emptyset \vdash \Lambda_{\mathcal{G}}(Y) : (v', \tau')$. If $v' = v \geq (2B)^k$, then we have an edge to a node from which there is a path having at least k productive edges, by the induction hypothesis. Otherwise $v > v' \geq (2B)^{k-1}$, so we have a productive edge to a node, from which there is a path having at least $k - 1$ productive edges, by the induction hypothesis.

Starting with $X = X_{\mathsf{st}}$ and $k > |\mathcal{T}_{\mathcal{G}}|$ we obtain a path from $(X_{\mathsf{st}} : (\mathsf{pr}, \mathsf{r}))$ containing more than $|\mathcal{T}_{\mathcal{G}}|$ productive edges; this path has to reach a cycle with a productive edge, as needed for Point (3). ◀

6.1 Implementation

Our implementation is based on the implementation of HorSat [4] (more precisely: of HorSat2, a revised version of the HorSat algorithm, due to Kobayashi and Terao). The main reason behind the efficiency of our (and HorSat's) implementation is that we do not compute all possible bindings in $\mathcal{T}_{\mathcal{G}}$. We derive only types which may be useful in a derivation of a type of the starting nonterminal X_{st}.

To this end, we first perform a 0-CFA analysis of the input to compute which lambda-terms may flow into particular nonterminal parameters. For example, if we have a nonterminal X with a parameter x, we find all applications in the form $M_X\, N$, where M_X is a lambda-term where the head is X or may eventually be substituted by X; then the lambda-term N is flowing into the parameter x. Note that 0-CFA analysis gives an overestimation. For example, for a nonterminal Y with two parameters x and y it is possible to write two applications, $Y\, M\, N$ and $Y\, M'\, N'$, transformed in such a way that, according to 0-CFA output, M may flow into x at the same time as N' into y. This behavior can exponentially (with respect to the number of arguments) increase the number of possible typings of Y that the algorithm has to check.

Thanks to 0-CFA analysis, we have a complete list of lambda-terms which may be substituted for parameters of nonterminals, which limits the set of computed types of nonterminals while still retaining all types required to type X_{st}. This is sufficient to start a loop where new types are found. In this loop, nonterminals (more precisely: the lambda-terms $\mathcal{R}(X)$ for nonterminals X) are typed using information about types flowing into their parameters. New types of nonterminals enable finding new types of lambda-terms in which these nonterminals occur. These lambda-terms again flow into parameters of some nonterminals, so the new types enable finding additional typings of these nonterminals, returning to the beginning of the loop. This loop continues until a fixpoint, that is, until no new typings of nonterminals or lambda-terms flowing into their parameters can be computed. As there is a finite number of possible typings of nonterminals and lambda-terms in any given recursion scheme, this fixpoint will be reached after finitely many steps. During this loop, each time we type a nonterminal, we take note of typings of nonterminals that were used in the derivation (i.e., are subtrees in the final derivation tree), and incrementally construct the derivation graph described earlier in this section. This way we find all parts of the derivation graph that are reachable from $X_{\mathsf{st}} : (\mathsf{pr}, \mathsf{r})$; the optimizations made by 0-CFA only remove unreachable parts.

We perform typing of terms without lambda-bindings in a top-down manner, that is, we iterate over all possible types an application may have, find compatible types for its left-hand side, and then type-check its right-hand side under each possible environment with its desired, already known type. The top-down approach is particularly efficient when the constant b is present in the term, since its argument with type $\top$ does not have to be type-checked. This is also the case for nonterminals with arguments of type $\top$.

We also include three optimizations similar to the ones present in HorSat:

- After a new type of a nonterminal is found, we type all lambda-terms that contain it in a way where the new typing is used at least once. This optimization can be turned off with a flag `-nofntty`.
- After a new type of a lambda-term flowing into a parameter x of a nonterminal X is found, we search for new typings of $\mathcal{R}(X)$ (or its subterms that flow into parameters of other nonterminals) in a way where at least one instance of the parameter x has the new type. This optimization can be turned off with a flag `-noftty`.
- When no parameter of a nonterminal X is used as a left-hand side of an application in $\mathcal{R}(X)$, we infer types of parameters of X from left-hand sides of applications instead of using types of lambda-terms flowing into these parameters. Then $\mathcal{R}(X)$ does not have to be typed whenever a new type is computed for lambda-terms flowing into its parameters. This optimization can be turned off with a flag `-nohvo`.

6.2 Benchmarks

We prepared benchmarks for the implementation of our algorithm by modifying benchmarks presented in the HorSat paper [4]. HorSat is an algorithm that efficiently checks whether an alternating tree automaton (ATA) accepts the Böhm tree of the lambda-term defined by a recursion scheme. This problem is different from ours, however, it also performs an analysis on the same trees and is also m-EXPTIME-complete, where the difficulty depends on the order m of the lambda-term. Hence, we use benchmark results presented in the HorSat paper [4] as an indication that the terms used there are difficult to analyze.

The details on the original benchmarks and their origin can be found in the HorSat paper [4]. According to authors of the aforementioned paper, these benchmarks contain practical data. Indeed, some of them model analysis of an XHTML document or a short computer program. Analysis of many of them was not completed in a reasonable time by other model checking algorithms similar to HorSat, while HorSat only failed to analyze benchmark `fibstring` in a reasonable time.

Table 1 InfSat benchmark results in all combinations of optimization flags.

		Flags and run times in seconds							
Benchmark name	$ord(\mathcal{G})$	no flags	`-noftty`	`-nofntty`	`-nohvo`	`-noftty` `-nofntty`	`-noftty` `-nohvo`	`-nofntty` `-nohvo`	`-noftty` `-nofntty` `-nohvo`
`jwig-cal_main`	2	0.942	0.930	1.572	0.611	1.574	0.607	0.728	0.783
`spec_cps_coerce1-c`	3	66.28	3.921	3.285	102.2	0.647	3.517	38.105	4.557
`xhtmlf-div-2`	2	9.591	9.555	7.457	9.238	7.426	9.007	7.387	7.478
`xhtmlf-m-church`	2	9.689	9.655	7.481	9.235	7.470	9.172	7.459	7.562
`fold_fun_list`	7	0.425	0.025	OOM	0.402	OOM	0.024	OOM	OOM
`fold_right`	5	25.01	0.017	0.277	23.75	0.028	0.016	0.265	0.028
`search-e-church`	6	127.7	14.96	338.1	119.5	12.19	14.11	325.9	13.53
`zip`	4	2.618	153.4	64.79	27.30	201.5	150.0	68.63	98.81
`filepath`	2	TO	OOM	OOM	OOM	OOM	OOM	OOM	OOM
`filter-nonzero-1`	5	59.69	1.635	37.56	64.19	2.636	1.641	39.52	1.922
`filter-nonzero`	5	0.641	0.106	0.570	0.670	0.138	0.106	0.653	0.083
`map-plusone-2`	5	5.384	1.082	2.591	5.065	0.851	1.092	5.633	1.125
`cfa-life2`	14	TO	TO	OOM	TO	TO	TO	TO	TO
`cfa-matrix-1`	8	1.794	2.376	1.672	1.708	2.124	2.310	1.582	2.280
`cfa-psdes`	7	0.017	0.022	0.019	0.026	0.022	0.018	0.041	0.029
`tak_inf`	8	10.48	5.842	12.16	9.759	11.88	5.739	11.58	8.858
`dna`	2	0.118	0.106	0.050	0.115	0.074	0.109	0.055	0.074
`fibstring`	4	0.022	0.024	0.016	0.023	0.020	0.023	0.015	0.020
`g45`	4	41.12	45.73	7.819	16.46	56.61	46.54	1.629	46.06
`l`	3	0.012	0.023	0.016	0.017	0.017	0.022	0.015	0.017
`fib02_odd_fin`	4	0.007	0.014	0.009	0.013	0.013	0.013	0.014	0.013
`fib_even_inf`	4	0.545	2.854	0.697	0.505	2.555	2.637	0.697	2.604
`two_add_inf`	4	0.022	0.018	0.023	0.024	0.015	0.012	0.021	0.015
`two_succ_inf`	4	0.005	0.006	0.005	0.006	0.004	0.005	0.006	0.005

TO means timeout (10 minutes), OOM means out of memory error.

Let us describe how we modified the benchmarks to fit our analysis. We left the original recursion scheme intact and selected a few important constants that are present close to leaves of generated trees to increase the difficulty. The result was a decision problem whether the operations described by important constants in modelled programs could be executed unbounded number of times, assuming the program halted. Additionally, we added four benchmarks specific to InfSat that answer mathematical questions such as whether there exist arbitrarily large odd numbers defined as Church numerals.

We present results of our benchmarks in Table 1. They were performed on a laptop with Intel Core i5-7600K, 16GB RAM. We consider these results a success, as only two benchmarks on practical data failed to compute within ten minutes. As InfSat is the first efficient algorithm solving the simultaneous unboundedness problem, we do not have any data to which we could compare our benchmark results. At this stage, we can assess effectiveness of our optimizations. We can see that using all optimization flags produced good results consistently, however, each optimization happened to slow down some benchmarks. The reason is that optimizations turned off by `-noftty` and `-nofntty` add substantial polynomial overhead when generating list of possible environments and optimization turned off by `-nohvo` changes the way types of some terms are computed. However, all of them can exponentially reduce the computation time in many cases which is why they are turned on by default.

7 Multi-letter case

It is not difficult to modify the type system to handle simultaneous unboundedness. In this part, instead of a single important constant a of arity 1, we consider important constants $\mathsf{a}_1, \dots, \mathsf{a}_s$, all of arity 1 ($\Sigma_\mathsf{a}$ is the set containing them all). Besides them, in the signature Σ we have a constant b of arity 2, and constants c and ω of arity 0.

In the type system instead of a single productivity flag in $\{\mathsf{pr}, \mathsf{np}\}$, we have a productivity set, being a subset of Σ_a, and saying which important letters are produced. Likewise, instead of a productivity value in $\mathbb{N}$, we have a productivity function $v\colon \Sigma_\mathsf{a} \to \mathbb{N}$, specifying a value separately for every letter. The rules of the type system are adopted in the expected way: $v(\mathsf{a}_i)$ increases when we use constant a_i, and when we duplicate a type binding for a type pair (A, σ) with $\mathsf{a}_i \in A$.

One more change is, however, necessary. Indeed, while proving Lemma 5.2 in the multi-letter case, we cannot choose a single subderivation concerning L that has the greatest value, since now a value is not a number, but rather a function. Instead, for every constant a_i we can choose a subderivation concerning L for which the i-th coordinate of the value is the greatest. We thus need to allow s (i.e., $|\Sigma_\mathsf{a}|$) subderivations for every type pair.

To this end, on the left of the arrow in a type, we do not have a set of types, but rather a multiset of types, where we allow to have at most s copies of every type. Likewise, in the type environment we have at most s copies of every type binding.

Formally, an *s-multiset* is a multiset that contains at most s copies of every element. The set of s-multisets of elements of X is denoted $\mathcal{P}_{\leq s}(X)$. Notice that 1-multisets are just sets, hence $\mathcal{P}_{\leq 1}(X) = \mathcal{P}(X)$. A union of s-multisets U, V, denoted $U \cup V$, is defined as follows: if U and V contain, respectively, n and m copies of an element x, then $U \cup V$ contains $\min(n+m, s)$ copies of this element. We use the notation $\{x_i \mid i \in I\}$ for $\bigcup_{i\in I}\{x_i\}$, where $\{x_i\}$ is the multiset containing x_i once.

For each sort α we define the set $\mathcal{T}_s^\alpha$ of types of sort α as follows:

$$\mathcal{T}_s^{\mathsf{o}} = \{\mathsf{r}\}, \qquad\qquad \mathcal{T}_s^{\alpha\to\beta} = \mathcal{P}_{\leq s}(\mathcal{P}(\Sigma_\mathsf{a}) \times \mathcal{T}_s^\alpha) \times \mathcal{T}_s^\beta.$$

We again use the notation with $\to$ and $\bigwedge$, but this time while writing $\bigwedge_{i\in I}(f_i, \tau_i) \to \tau$, we assume that any pair occurs as (f_i, τ_i) at most s times.

A *type judgment* is of the form $\Gamma \vdash M : (v, \tau)$, where $v\colon \Sigma_\mathsf{a} \to \mathbb{N}$, and where we require that the type τ and the lambda-term M are of the same sort. The *type environment* Γ is an s-multiset of bindings of variables of the form $x^\alpha : (A, \tau)$, where $A \subseteq \Sigma_\mathsf{a}$ is a *productivity set* and $\tau \in \mathcal{T}_s^\alpha$ is a type. For $a \in \Sigma_\mathsf{a}$ by $\Gamma{\restriction_a}$ we denote the s-multiset of those binding from Γ that have a in their productivity set.

By $\mathbf{0}$ we denote the function from Σ_a to $\mathbb{N}$ that maps every a_i to 0, and by χ_i the function that maps a_i to 1 and all other a_j to 0. The type system consists of the following rules:

$$\emptyset \vdash \mathsf{a}_i : (\chi_i, (A, \mathsf{r}) \to \mathsf{r}) \qquad\qquad \emptyset \vdash \mathsf{c} : (\mathbf{0}, \mathsf{r})$$

$$\emptyset \vdash \mathsf{b} : (\mathbf{0}, (A, \mathsf{r}) \to \top \to \mathsf{r}) \qquad \emptyset \vdash \mathsf{b} : (\mathbf{0}, \top \to (A, \mathsf{r}) \to \mathsf{r}) \qquad x : (A, \tau) \vdash x : (\mathbf{0}, \tau)$$

$$\frac{\Gamma \cup \{x : (A_i, \tau_i) \mid i \in I\} \vdash K : (v, \tau) \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.K : (v, \bigwedge_{i\in I}(A_i, \tau_i) \to \tau)}\ (\lambda)$$

We define the duplication factor $dupl((\Gamma_j)_{j\in J})$, as the function (from Σ_{a} to $\mathbb{N}$) that maps every important constant $a \in \Sigma_{\mathsf{a}}$ to $\sum_{j\in J} |\Gamma_j\restriction_a| - \left|\bigcup_{j\in J} \Gamma_j\restriction_a\right|$.

$$\frac{\begin{array}{c}0 \notin I \qquad \forall i \in I.\ A_i = \{a \in \Sigma_{\mathsf{a}} \mid v_i(a) > 0 \vee \Gamma_i\restriction_a \neq \emptyset\} \\ \Gamma_0 \vdash K : (v_0, \bigwedge_{i\in I}(A_i, \tau_i) \to \tau) \qquad \Gamma_i \vdash L : (v_i, \tau_i) \text{ for each } i \in I\end{array}}{\bigcup_{i\in\{0\}\cup I} \Gamma_i \vdash K\,L : (dupl((\Gamma_i)_{i\in\{0\}\cup I}) + \sum_{i\in\{0\}\cup I} v_i, \tau)}\ (@)$$

Recall that this time any pair may occur as (f_i, τ_i) at most s times.

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 William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logical Methods in Computer Science*, 5(1), 2009. doi:10.2168/LMCS-5(1:3)2009.

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 Arnaud Carayol and Olivier Serre. Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 165–174. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.73.

6 Lorenzo Clemente, Paweł 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.

7 Lorenzo Clemente, Paweł 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.

8 Wojciech Czerwiński, Wim Martens, Lorijn van Rooijen, Marc Zeitoun, and Georg Zetzsche. A characterization for decidable separability by piecewise testable languages. *Discrete Mathematics & Theoretical Computer Science*, 19(4), 2017. doi:10.23638/DMTCS-19-4-1.

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

10 Matthew Hague, Jonathan Kochems, and C.-H. Luke Ong. Unboundedness and downward closures of higher-order pushdown automata. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 151–163. ACM, 2016. doi:10.1145/2837614.2837627.

11 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. *ACM Trans. Comput. Log.*, 18(3):25:1–25:42, 2017. doi:10.1145/3091122.

12 Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. Higher-order pushdown trees are easy. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002. `doi:10.1007/3-540-45931-6_15`.

13 Naoki Kobayashi. Model checking higher-order programs. *J. ACM*, 60(3):20:1–20:62, 2013. `doi:10.1145/2487241.2487246`.

14 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`.

15 Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233. ACM, 2011. `doi:10.1145/1993498.1993525`.

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 Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. General decidability results for asynchronous shared-memory programs: Higher-order and beyond. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 449–467. Springer, 2021. `doi:10.1007/978-3-030-72016-2_24`.

18 Robin P. Neatherway and C.-H. Luke Ong. TravMC2: higher-order model checking for alternating parity tree automata. In Neha Rungta and Oksana Tkachuk, editors, *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*, pages 129–132. ACM, 2014. `doi:10.1145/2632362.2632381`.

19 Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 353–364. ACM, 2012. `doi:10.1145/2364527.2364578`.

20 C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 81–90. IEEE Computer Society, 2006. `doi:10.1109/LICS.2006.38`.

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

22 Paweł 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`.

23 Paweł Parys. Homogeneity without loss of generality. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.FSCD.2018.27`.

24 Paweł Parys. Intersection types for unboundedness problems. In Michele Pagani and Sandra Alves, editors, *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems, DCM/ITRS 2018, Oxford, UK, 8th July 2018*, volume 293 of *EPTCS*, pages 7–27, 2018. `doi:10.4204/EPTCS.293.2`.

25 Paweł Parys. Recursion schemes, the MSO logic, and the U quantifier. *Log. Methods Comput. Sci.*, 16(1), 2020. `doi:10.23638/LMCS-16(1:20)2020`.

26 Paweł Parys. A type system describing unboundedness. *Discret. Math. Theor. Comput. Sci.*, 22(4), 2020. doi:10.23638/DMTCS-22-4-2.

27 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.

28 Hiroshi Unno, Naoshi Tabuchi, and Naoki Kobayashi. Verification of tree-processing programs via higher-order mode checking. *Mathematical Structures in Computer Science*, 25(4):841–866, 2015. doi:10.1017/S0960129513000054.

29 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.