Linearity in Higher-Order Recursion Schemes

PIERRE CLAIRAMBAULT, Univ Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, Lyon, France
CHARLES GRELLOIS, Inria, Sophia Antipolis and Aix Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296, Marseille, France
ANDRZEJ S. MURAWSKI, University of Oxford, United Kingdom

Higher-order recursion schemes (HORS) have recently emerged as a promising foundation for higher-order program verification. We examine the impact of enriching HORS with linear types. To that end, we introduce two frameworks that blend non-linear and linear types: a variant of the λY-calculus and an extension of HORS, called linear HORS (LHORS).

First we prove that the two formalisms are equivalent and there exist polynomial-time translations between them. Then, in order to support model-checking of (trees generated by) LHORS, we propose a refined version of alternating parity tree automata, called LNAPTA, whose behaviour depends on information about linearity. We show that the complexity of LNAPTA model-checking for LHORS depends on two type-theoretic parameters: linear order and linear depth. The former is in general smaller than the standard notion of order and ignores linear function spaces. In contrast, the latter measures the depth of linear clusters inside a type. Our main result states that LNAPTA model-checking of LHORS of linear order n is n-EXPTIME-complete, when linear depth is fixed. This generalizes and improves upon the classic result of Ong, which relies on the standard notion of order.

To illustrate the significance of the result, we consider two applications: the MSO model-checking problem on variants of HORS with case distinction (RSFD and HORSC) on a finite domain and a call-by-value resource verification problem. In both cases, decidability can be established by translation into HORS, but the implied complexity bounds will be suboptimal due to increases in type order. In contrast, we show that the complexity bounds derived by translations into LHORS and appealing to our result are optimal in that they match the respective hardness results.

CCS Concepts: • **Software and its engineering → Model checking**; • **Theory of computation →** *Linear logic*;

Additional Key Words and Phrases: Higher-order computation, recursion schemes, linear logic, model checking

ACM Reference Format:
Pierre Clairambault, Charles Grellois, and Andrzej S. Murawski. 2018. Linearity in Higher-Order Recursion Schemes. *Proc. ACM Program. Lang.* 2, POPL, Article 39 (January 2018), 29 pages. https://doi.org/10.1145/3158127

1 INTRODUCTION

Higher-order recursion schemes (HORS) are typed grammars that generate potentially infinite ranked trees. Their history can be tracked back to early research in semantics in the 1970s [Damm 1977; Nivat 1972], but recently they have become a successful foundation for program verification, in the functional paradigm [Kobayashi 2009] and beyond [Tsukada and Kobayashi 2010].

Authors' addresses: Pierre Clairambault, Univ Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP, Lyon, France, pierre.clairambault@ens-lyon.fr; Charles Grellois, Inria, Sophia Antipolis and Aix Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296, Marseille, France, charles.grellois@inria.fr; Andrzej S. Murawski, Department of Computer Science, University of Oxford, United Kingdom.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART39
https://doi.org/10.1145/3158127

At the theoretical level, the approach takes advantage of the fact that monadic second-order logic (MSO) over trees generated by HORS is decidable [Ong 2006]. In practice, though, various restricted versions of the problem are being considered [Kobayashi 2009; Kobayashi and Ong 2011], notably reachability. Numerous verification tools, e.g. [Broadbent et al. 2013; Kobayashi et al. 2011; Murase et al. 2016; Ramsay et al. 2014], have been constructed to test the approach, exploring both type-based and automata-theoretic techniques. Krivine machines have also been exploited in that context [Salvati and Walukiewicz 2016] as well as finite models [Aehlig 2007; Grellois and Melliès 2015a; Hofmann and Ledent 2017; Salvati and Walukiewicz 2015]. Despite the rather disappointing worst-case complexity (n-EXPTIME for MSO model-checking of order-n HORS), experiments paint a more promising story and there is need for identifying better complexity indicators than standard type-theoretic order.

In order to adapt the original results on HORS to more advanced scenarios, it is often necessary to encode desirable features or evaluation mechanisms into the rather crude language of (call-by-name) HORS. This often leads to increases in type order, e.g. due to CPS transformations, suggesting drastic changes to expected worst-case complexity. In this paper we introduce *linear higher-order recursion schemes* (LHORS) that facilitate a more refined type-theoretic analysis and, consequently, make it possible to derive more accurate complexity bounds. This is achieved by mixing linear and non-linear types [Girard 1987]. In order to model-check such schemes, we introduce a special notion of *linear non-linear alternating parity automaton* (LNAPTA), which are a refinement of alternating parity automata that is sensitive to linearity.

We find that the complexity of LNAPTA model-checking for LHORS is governed by two crucial parameters: *linear order* and *linear depth*. In particular, the linear order of a type remains unaffected by linear function spaces and, consequently, can be much smaller than the corresponding standard order of a type. Linear depth, in turn, measures the depth of linear clusters inside a type (by a linear cluster we mean, roughly, a segment of contiguous linear type constructors, separated by non-linear arrows). Our main result states that LNAPTA model-checking of LHORS of *linear* order n is n-EXPTIME-complete, when linear depth is fixed. This subsumes the result of Ong [2006], as the linear depth of any intuitionistic (non-linear) type is always equal to 0.

Our proof extends the intersection-type approach of Kobayashi [2009] to linear and product types, and takes advantage of the observation that, when it comes to counting type refinements, standard function spaces induce exponential blow-ups in the search space while linear ones do not.

HORS are well known to be equivalent in expressivity to Böhm trees of ground-type λY-calculus terms, under the assumption that their free variables are of at most order 1, which makes it possible to represent leaves and nodes [Salvati and Walukiewicz 2016]. Accordingly, we start our analysis by introducing a calculus, called the $\lambda\!\!\lambda Y$-calculus, that combines the expressivity of λY with a linear type system. The $\lambda\!\!\lambda Y$-calculus will turn out convenient as the target language of translations that aim to take advantage of our type system. To match the calculus, we define *linear higher-order recursion schemes* (LHORS) and show that they induce the same trees as the $\lambda\!\!\lambda Y$-calculus in a way that preserves linear order and depth. The translations in both directions are of polynomial-time complexity, meaning that the two formalisms can be used interchangeably for the purpose of calculating complexity bounds.

Using the new developments, one can study model-checking problems for formalisms different and richer than HORS by translation into LHORS. We demonstrate this by generalizing several results from the literature. Without the refined analysis afforded by our result, the relevant translations would imply exponential or doubly-exponential cost. In contrast, the complexity bounds that we can derive by translation are optimal and match the respective hardness results.

As applications of the new result, we first revisit two other formalisms used in higher-order model-checking: *recursive schemes over finite data domains* (RSFD) [Kobayashi et al. 2010] and *higher-order recursion schemes with cases* (HORSC) [Neatherway et al. 2012]. In both cases, we show how one can translate the associated terms into LHORS in such a way that the linear order will be unaffected by the translation, even though the standard type-theoretic order would. The linear depth will grow during the translation, but only by a constant. This makes it possible to extend Ong's result for order-n HORS [Ong 2006] (i.e. n-EXPTIME-completeness of MSO model-checking) to RSFD and HORSC simply by translation.

Finally, we investigate a call-by-value framework. Here the associated reachability problem was shown to be n-EXPTIME-complete for programs of depth-n in Tsukada and Kobayashi [2014]. We consider the more general resource usage verification problem and also show its n-EXPTIME-completeness. In contrast to Tsukada and Kobayashi [2014], our result is obtained through a linear variant of a CPS transformation, which maps call-by-value programs of depth n to call-by-name $\lambda\!\!\lambda Y$-terms of linear order n. CPS translations are known to cause an increase in type order, but in our case *linear order* does not increase. As the linear depth of the types involved turns out to be constant (equal to 2), our result for LHORS implies the desired complexity.

To sum up, we present a new unifying framework, founded on linear types. The addition of linearity is shown to have far-reaching consequences:

(1) our results subsume the original ones on HORS for non-linear types,
(2) they provide much more accurate complexity-theoretic bounds than any earlier work,
(3) many existing results can be unified and extended simply by translating into the framework.

Consequently, we believe our framework to be highly suitable as a metalanguage for future work in higher-order verification.

2 THE $\lambda\!\!\lambda Y$-CALCULUS

We start off by introducing the $\lambda\!\!\lambda Y$-calculus, which is a simply-typed λ-calculus extended with a fixpoint combinator (in the spirit of the λY-calculus [Statman 2004]) and refined with a linear-non-linear type system (in the style of [Barber and Plotkin 1996]). We refer to its types as *kinds* to avoid collision with the intersection type system to come.

2.1 Kinds and Terms

2.1.1 Kinds and Their Measures. The *kinds* include a ground kind o (the kind of *trees*), two arrow constructors $\rightarrow$ (standard non-linear arrow) and $\multimap$ (linear arrow [Girard 1987]), and products &. Furthermore, we define the kinds as having the following restricted shape.

Definition 1. **Kinds** are generated by either of φ or ϖ in the following grammar.

$$\begin{array}{rcl} \varphi, \psi, \ldots & ::= & o \mid \varpi \multimap \psi \mid \varphi \rightarrow \psi \\ \varpi, \kappa, \iota, \ldots & ::= & \varphi \mid \&_{i \in I} \varphi_i \end{array}$$

with I any finite set – we write $\varphi \& \psi$ for the binary case.

We refer to kinds generated by φ as *functional* kinds, and those generated by ϖ as *product* kinds. Note that any functional kind can be regarded as a (singleton) product kind, so $\varpi, \kappa, \iota, \ldots$ really range over arbitrary kinds. Abusing notation, we sometimes identify a functional kind φ and the unary product $\&_{\{\star\}}\varphi$. This allows us to write expressions such as $\&_{i \in I}\varphi_i \rightsquigarrow \psi$, where $\rightsquigarrow$ is either $\multimap$ or $\rightarrow$ (it is then understood that I is singleton if $\rightsquigarrow = \rightarrow$).

Our restriction on kinds amounts to forbidding products on the left of a non-linear arrow, and on the right of any arrow. However, any non-restricted kind built from $\{o, \multimap, \to, \&\}$ is *isomorphic* to a kind from Definition 1 through the isomorphisms (of Intuitionistic Linear Logic [Girard 1987]):

$$\kappa \to (\&_{i\in I}\kappa_i) \;\cong\; \&_{i\in I}(\kappa \to \kappa_i) \qquad\qquad \kappa \multimap (\&_{i\in I}\kappa_i) \;\cong\; \&_{i\in I}(\kappa \multimap \kappa_i)$$

$$(\&_{i\in\{1,\ldots,n\}}\kappa_i) \to \kappa \;\cong\; \kappa_1 \to \cdots \to \kappa_n \to \kappa$$

The first two equations hide a combinatorial explosion: converting a non-restricted kind to one in Definition 1 may cause an exponential blow-up. For some of our developments, it will be important that we opt for the notion of kind in Definition 1, with no such hidden complexity.

Measures. The **size** $|\kappa|$ and **linear order** $\ell o(\kappa)$ of a kind κ are defined inductively:

$$\begin{array}{rclcrcl} |o| & = & 1 & & \ell o(o) & = & 0 \\ |\varpi \multimap \varphi| & = & |\varpi| + |\varphi| + 1 & & \ell o(\varpi \multimap \varphi) & = & \max(\ell o(\varpi), \ell o(\varphi)) \\ |\varphi \to \psi| & = & |\varphi| + |\psi| + 1 & & \ell o(\varphi \to \psi) & = & \max(\ell o(\varphi) + 1, \ell o(\psi)) \\ |\&_{i\in I}\,\varphi_i| & = & 1 + \sum_{i\in I}|\varphi_i| & & \ell o(\&_{i\in I}\varphi_i) & = & \max_{i\in I}\ell o(\varphi_i) \end{array}$$

While the definition of size is straightforward, that of linear order deserves a comment. If we ignore the clauses for $\multimap$ and $\&$, we simply obtain the standard notion of order of a simple type, see e.g. [Salvati and Walukiewicz 2016]. The order is extended to $\multimap$ and $\&$ simply by ignoring the constructors and taking the maximum of the linear order of their operands. This should be natural to the reader familiar with *linear logic* [Girard 1987]: indeed, recall that the *call-by-name Girard translation* interprets $\varphi \to \psi$ as $!\varphi \multimap \psi$ in linear logic. Hence, one can understand the definition of linear order above as simply counting the maximal nesting of exponentials in the corresponding linear logic formula. A key inspiration for this work is the idea that the main source of complexity in the higher-order model-checking problem is the exponential, and not the arrow in itself.

While our main theorem will substantiate this claim, there will be a proviso because linear kinds, to an extent, *do* impact complexity. Consequently, to arrive at our asymptotic bounds, we will need to control the use of linear kinds by bounding the extent to which purely linear type constructors can occur contiguously (without being separated by the left-hand side of a non-linear arrow, *i.e.* by a !). For that purpose we introduce the notion of *linear depth* of a kind.

Definition 2. The **local linear depth** $\mathrm{lld}(\kappa)$ of a kind κ is defined inductively as follows

$$\begin{array}{rclcrcl} \mathrm{lld}(o) & = & 0 & & \mathrm{lld}(\varpi \multimap \varphi) & = & \max(\mathrm{lld}(\varpi), \mathrm{lld}(\varphi)) + 1 \\ \mathrm{lld}(\&_{i\in I}\varphi_i) & = & \max_{i\in I}\mathrm{lld}(\varphi_i) + 1 & & \mathrm{lld}(\varphi \to \psi) & = & \mathrm{lld}(\psi) \end{array}$$

The **linear depth** of κ, written $\ell d(\kappa)$, is the maximum of $\mathrm{lld}(\iota)$, taken over all subkinds ι of κ.

Observe that purely non-linear kinds have linear depth 0. We shall see that, even though bounded linear depth increases expressivity, this does not affect the asymptotic complexity of model-checking. In particular, our translations of Sections 5 and 6 will yield kinds with bounded linear depth.

2.1.2 Kinded Terms and Their Measures.

Definition 3. **Raw terms** are defined as follows.

$$t, u \quad ::= \quad x \mid \lambda x^{\varphi}.\, u \mid \ell x^{\kappa}.\, u \mid t\, u \mid \langle u_i \mid 1 \le i \le n\rangle \mid \pi_i \mid \mathbb{Y}x^{\kappa}.\, u$$

After Definition 1, we wrote that we will sometimes identify a functional kind φ with the unary product $\&_{\{\star\}}\varphi$. Accordingly, we will also identify a (non-product) term t and the unary tuple $\langle t \mid i \in \{\star\}\rangle$ – this will enable more uniform notations later on.

$$\frac{}{\Gamma, x :: \kappa \mid \Delta \vdash x :: \kappa} \qquad \frac{j \in I}{\Gamma \mid \Delta, x :: \&_{i\in I}\, \varphi_i \vdash \pi_j\, x :: \varphi_j}$$

$$\frac{\Gamma \mid \Delta_1 \vdash t :: \kappa \multimap \varphi \qquad \Gamma \mid \Delta_2 \vdash u :: \kappa}{\Gamma \mid \Delta_1, \Delta_2 \vdash t\, u :: \varphi} \qquad \frac{\Gamma \mid \Delta \vdash t :: \varphi \to \psi \qquad \Gamma \mid _ \vdash u :: \varphi}{\Gamma \mid \Delta \vdash t\, u :: \psi}$$

$$\frac{\Gamma, x :: \varphi \mid \Delta \vdash t :: \psi}{\Gamma \mid \Delta \vdash \lambda x^{\varphi}.\, t :: \varphi \to \psi} \qquad \frac{\Gamma \mid \Delta, x :: \kappa \vdash t :: \varphi}{\Gamma \mid \Delta \vdash \ell x^{\kappa}.\, t :: \kappa \multimap \varphi} \qquad \frac{\Gamma \mid \Delta \vdash t_i :: \varphi_i \qquad (1 \le i \le n)}{\Gamma \mid \Delta \vdash \langle t_i \mid 1 \le i \le n \rangle :: \&_{1\le i\le n}\varphi_i}$$

$$\frac{\Gamma \mid \Delta \vdash t :: \&_{1\le i\le n}\varphi_i}{\Gamma \mid \Delta \vdash \pi_i\, t :: \varphi_i} \qquad \frac{\Gamma, x : \kappa \mid _ \vdash t :: \kappa}{\Gamma \mid _ \vdash \mathbb{Y} x^{\kappa}.\, t :: \kappa}$$

Fig. 1. Kinding rules for $\lambda\!\ell Y$

Our abstractions explicitly carry kinds: the language is *à la Church*. For notational convenience we have two different function abstractions: one for the linear arrow (ℓ) and another for the non-linear arrow (λ). This is a superficial distinction: as in this paper we will only consider well-kinded terms, the information of whether an abstraction is linear or not is redundant with the kinds.

Now, we give the kinding rules for terms. The **kinding judgements** of $\lambda\!\ell Y$ have the form $\Gamma \mid \Delta \vdash u :: \kappa$, where $\Gamma = x_1 :: \kappa_1, \ldots, x_n :: \kappa_n$ is a **non-linear context** and $\Delta = y_1 :: \kappa_1, \ldots, y_p :: \kappa_p$ is a **linear context**. The rules are given in Figure 1. The non-linear context may comprise variables of product kinds – though these cannot be abstracted, they can be used to compute a fixpoint. Empty contexts are denoted as _. In particular, note that $\lambda\!\ell Y$ contains the standard λY-calculus [Statman 2004] as a sub-language: any λY-term $\Gamma \vdash t : \varphi$ can be kinded in $\lambda\!\ell Y$ with $\Gamma \mid _ \vdash t :: \varphi$.

From now on, all terms are implicitly kinded. We write $\mathsf{KT}_{\Gamma|\Delta}(\kappa)$ for the set of terms $\Gamma \mid \Delta \vdash t :: \kappa$. Every subterm of a kinded term t automatically comes with a kind, so we can say whether a subterm $u_1\, u_2$ of t is a *linear* application (if u_1 has kind $\varpi \multimap \varphi$) or a *non-linear* one (if u_1 has kind $\varphi \to \psi$).

Measures. The **linear order** of $t \in \mathsf{KT}_{\Gamma|\Delta}(\kappa)$, written $\ell o(t)$, is the maximum of all $\ell o(\kappa)$, where κ ranges over the kinds of subterms of t. The **local linear depth** of t (written $\mathrm{lld}(t)$) is defined to be

$$\mathrm{lld}(\varpi_1 \multimap \ldots \multimap \varpi_n \multimap \kappa)$$

where $\Delta = x_1 :: \varpi_1, \ldots, x_n :: \varpi_n$. The **linear depth** $\ell d(t)$ of t is taken to be the maximum of all $\mathrm{lld}(u)$, where u ranges over subterms of t. We also define the **size** of t, written $|t|$, as follows.

$$\begin{array}{rclrcl} |x| &=& 1 & |t_1\, t_2| &=& |t_1| + |t_2| + 1 \\ |\lambda x^{\kappa}.\, t| &=& |\kappa| + |t| + 1 & |\ell x^{\kappa}.\, t| &=& |\kappa| + |t| + 1 \\ |\pi_i\, t| &=& |t| + 1 & |\langle t_i \mid i \in I \rangle| &=& 1 + \sum_{i\in I} |t_i| \\ |\mathbb{Y} x^{\kappa}.\, t| &=& |\kappa| + |t| + 1 & & & \end{array}$$

Representing trees. In the λY-calculus or Higher-Order Recursion Schemes, trees are usually represented as normal terms of kind o, using variables of first-order kind. For instance, a variable $\mathrm{b} : \underbrace{o \to \cdots \to o}_{n} \to o$ can be used to represent n-ary branching, i.e. tree nodes with n descendants. Let Σ be a finite set of such variables. Salvati and Walukiewicz [2016] have shown that *Böhm trees* of λY-terms of the form $\Sigma \vdash u : o$ coincide with the trees generated by order-n recursion schemes, where Σ is taken to be the set of *terminal symbols* of the scheme.

In $\lambda\!\ell Y$, there is more flexibility as to how branching may be kinded. The role of the *first-order kinds* of λY (or HORS) will be played by the more general notion of *tree kinds*, defined below, which

$$\begin{array}{rclrclr} C[(\lambda x^{\varphi}.\,t)\,u] & \triangleright_\beta & C[t[u/x]] & C[\lambda x^{\varphi}.\,t\,x] & \triangleright_\eta & C[t] & x\notin \mathrm{fv}(t) \\ C[(\ell x^{\kappa}.\,t)\,u] & \triangleright_\beta & C[t[u/x]] & C[\ell x^{\kappa}.\,t\,x] & \triangleright_\eta & C[t] & x\notin \mathrm{fv}(t) \\ C[\pi_i\,\langle t_i \mid i\in I\rangle] & \triangleright_\beta & C[t_i] & C[\langle \pi_i\,t \mid i\in I\rangle] & \triangleright_\eta & C[t] & \end{array}$$

Fig. 2. β-reduction and η-contraction on $\lambda\!\!\!\ell Y$-terms

may have linear order 0 or 1. We shall see that the choice of kinding may have significant impact on the set of infinite trees generated and the set of properties that can be verified.

Definition 4. **Tree kinds** are the kinds θ generated by

$$\theta \quad ::= \quad o \mid \nu \multimap \theta \mid o \to \theta \qquad \text{and} \qquad \nu \quad ::= \quad \&_{1\le i\le n}\, o$$

A **tree signature** is a finite list $\Sigma = \mathrm{b}_1 :: \theta_1, \ldots, \mathrm{b}_n :: \theta_n$ where θ_i is a tree kind for all $1 \le i \le n$.

Remark 5. Tree kinds have the form $\theta = \&_{1\le j\le p_1} o \rightsquigarrow_1 \ldots \rightsquigarrow_{n-1} \&_{1\le j\le p_n} o \rightsquigarrow_n o$, where $\rightsquigarrow_i \in \{\to, \multimap\}$, and $p_i = 1$ whenever $\rightsquigarrow_i = \to$, with the convention that o may be written as $\&_{\{\star\}}\, o$ to uniformize notations. A tree kind θ induces a first-order kind $\{\!|\theta|\!\}$ if one ignores the linear information: $\{\!|\theta|\!\} = \underbrace{o \to \cdots \to o}_{p_1} \to \cdots \to \underbrace{o \to \cdots \to o}_{p_n} \to o$ is called the *delinearization* of θ. Likewise, by applying the above transformation to each kind of a tree signature Σ, we can obtain a first-order signature in the standard sense, written $\{\!|\Sigma|\!\}$.

If Σ is a tree signature, any finite tree on $\{\!|\Sigma|\!\}$ can be presented as a finite term $\Sigma \mid _ \vdash t :: o$, the latter possibly containing tuples to represent different branches. For instance, the tree of Example 11 (with dots replaced with $\bot :: o$) is represented by the term $t \in \mathrm{KT}_{\Gamma|_}(o)$ given below on the left with $\Gamma = \mathrm{b} :: o \,\&\, o \multimap o, \mathrm{c} :: o \multimap o, \mathrm{d} :: o \multimap o, \mathrm{e} :: o, \bot :: o$. Note that tuples of terms were introduced to match the linear specification of Σ. Next we define notions of reduction on $\lambda\!\!\!\ell Y$-terms, and explain how to use them to generate infinite trees represented in an analogous way.

$$\begin{array}{l} \mathrm{b}\,\langle \mathrm{c}\,(\mathrm{d}\,\mathrm{e}), \\ \quad \mathrm{b}\,\langle \mathrm{c}\,(\mathrm{c}\,(\mathrm{d}\,(\mathrm{d}\,\mathrm{e}))), \\ \quad\quad \mathrm{b}\,\langle \mathrm{c}\,(\mathrm{c}\,(\mathrm{c}\,(\mathrm{d}\,(\mathrm{d}\,(\mathrm{d}\,\mathrm{e}))))), \\ \quad\quad\quad \mathrm{b}\,\langle \mathrm{c}\,(\mathrm{c}\,(\mathrm{c}\,(\mathrm{c}\,\bot))), \bot\rangle\rangle\rangle\rangle \end{array}$$

2.2 Reduction and Böhm Trees

In this section, we study operational properties of the $\lambda\!\!\!\ell Y$-calculus. We define notions of reduction, state its basic properties, and define the *Böhm tree* (infinite normal form) of a term.

2.2.1 β-reduction and η-contraction. We define as usual a **context** as a term with a hole, *i.e.* a term defined by the following grammar:

$$\begin{array}{rcl} C[-] & ::= & [-] \mid \lambda x^{\varphi}.\,C[-] \mid \ell x^{\kappa}.\,C[-] \mid C[-]\,u \mid t\,C[-] \mid \pi_i\,C[-] \mid \\ & & \langle u_1, \ldots, u_{i-1}, C[-], u_{i+1}, \ldots, u_n\rangle \mid \mathbb{Y} x^{\kappa}.\,C[-] \end{array}$$

The basic reductions are β-**reduction** (for linear and non-linear functions, and for products), and η-**contraction** (again, for all three constructors). The rules are in Figure 2 – $\mathrm{fv}(t)$ denotes the set of free variables in t, either linear or non-linear. We will occasionally refer to η-**expansion**, the opposite of η-contraction, only defined for terms of appropriate kind. We write $\triangleright_{\beta\eta}$ for the union of $\triangleright_\beta, \triangleright_\eta$. Without any rules for unfolding fixpoints, reduction is strongly normalizing:

Proposition 6. *The reduction $\triangleright_{\beta\eta}$ is confluent, and strongly normalizing.*

Proof. Immediate by embedding into the simply-typed λ-calculus with surjective pairing, which is well-known to be strongly normalizing [Pottinger 1981]. ◻

2.2.2 Unfolding Fixpoints to Böhm Trees. For simplicity, when computing the Böhm tree of a term we will only expand fixpoints in *head position*. We define the **head contexts** as follows.

$$H[-] ::= [-] \mid \lambda x^{\varphi}.\, H[-] \mid \ell x^{\kappa}.\, H[-] \mid H[-]\, u \mid \pi_i\, H[-]$$

The fixpoint expansion rule is then defined to be

$$H[\mathbb{Y}x^{\kappa}.\, t] \triangleright_{\delta} H[t[\mathbb{Y}x^{\kappa}.\, t/x]].$$

Let us write $\triangleright_{\beta\eta\delta}$ for the union of $\triangleright_{\beta\eta}$ and $\triangleright_{\delta}$. Clearly, $\triangleright_{\delta}$ and $\triangleright_{\beta\eta\delta}$ are no longer normalizing: for instance, we have $\mathbb{Y}x^{o}.\, x \triangleright_{\delta} \mathbb{Y}x^{o}.\, x$. However, we retain confluence.

Lemma 7. *The reduction $\triangleright_{\beta\eta\delta}$ is confluent.*

Proof. Elementary reasoning, using determinism of $\triangleright_{\delta}$ (by construction there is at most one $\triangleright_{\delta}$-redex in a term), confluence of $\triangleright_{\beta\eta}$ and easy commutations properties between $\triangleright_{\delta}$ and $\triangleright_{\beta\eta}$. □

While $\triangleright_{\beta\eta\delta}$ does not necessarily terminate, it will produce a **head normal form** (*i.e.* a $\triangleright_{\beta\eta}$-normal term of the form $H[x]$ for a variable x) whenever it does. If $\triangleright_{\beta\eta\delta}$ terminates on t, we also say that it is **solvable**, following standard terminology in λ-calculus. We are now in position to define the **Böhm tree** of a kinded term.

Definition 8. Let $\Gamma \mid \Delta \vdash t :: \kappa$ be a kinded term. Its **Böhm tree**, written $\mathrm{BT}(t)$, is defined (co)recursively as follows.

$$\begin{array}{lcll}
\mathrm{BT}(t) & = & \langle \mathrm{BT}(\pi_i\, t) \mid i \in I \rangle & (\kappa = \&_{i \in I} \varphi_i) \\
\mathrm{BT}(t) & = & \lambda x^{\varphi}.\, \mathrm{BT}(t\, x) & (\kappa = \varphi \to \psi) \\
\mathrm{BT}(t) & = & \ell x^{\varpi}.\, \mathrm{BT}(t\, x) & (\kappa = \varpi \multimap \psi) \\
\mathrm{BT}(t) & = & \Omega & (\kappa = o \text{ and } t \text{ is not solvable}) \\
\mathrm{BT}(t) & = & x\, \mathrm{BT}(t_1)\, \ldots\, \mathrm{BT}(t_n) & (\kappa = o \text{ and } t \triangleright_{\beta\eta\delta} x\, t_1\, \ldots\, t_n)
\end{array}$$

Böhm trees give a notion of infinite normal forms for $\lambda\ell Y$-terms. In this paper, we primarily use it to define the infinite tree generated by a $\lambda\ell Y$-term of ground kind with a tree signature.

2.2.3 Infinite Trees Generated by $\lambda\ell Y$-terms. As hinted in Section 2.1.2, all finite $\triangleright_{\beta\eta\delta}$-normal terms $\Sigma \mid _ \vdash t :: o$ with Σ a tree signature are representations of finite trees on $\{\!|\Sigma|\!\}$. This motivates:

Definition 9. For any $\Sigma \mid _ \vdash t :: o$, the **infinite tree generated by** t is defined as $\mathrm{BT}(t)$.

Let us identify tuples with branching and view, for instance, a tree $\mathrm{b}\,\langle T_1, \ldots\, T_n \rangle$ as equivalent to the tree $\mathrm{b}\, T_1 \cdots T_n$ (such transformations will be formalized later as *delinearizations* of trees). Then, an infinite tree can be generated by a $\lambda\ell Y$-term iff it can be generated by a λY-term, or a HORS – it is straightforward to establish this by providing translations in both directions. However, the trees generated are different if one takes the *(linear) order* into account. Let us write BT^{ℓ}_n for the set of (possibly infinite) trees generated by a $\lambda\ell Y$-term of linear order n. We will write BT_n for the set of (possibly infinite) trees generated by a λY-term of order n. By [Salvati and Walukiewicz 2016], BT_n is equal to the set of trees generated by higher-order recursion schemes of order n. Because any λY-term of order n is also a $\lambda\ell Y$-term of linear order n, we can immediately conclude the following:

Proposition 10. *For all n, we have* $\mathrm{BT}_n \subseteq \mathrm{BT}^{\ell}_n$.

However, the other inclusion fails. In particular, we have $\mathrm{BT}^{\ell}_0 \not\subseteq \mathrm{BT}_1$.

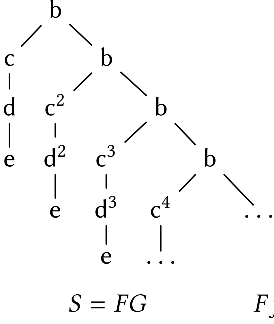
Example 11. Consider the tree signature $\Sigma = \{\mathrm{b} :: o \,\&\, o \multimap o, \mathrm{c} :: o \multimap o, \mathrm{d} :: o \multimap o, \mathrm{e} :: o\}$. Consider the $\lambda\ell Y$-term t given below.

$$\Sigma \mid _ \vdash (\mathbb{Y}F^{(o \multimap o) \multimap o}.\, \ell f^{o \multimap o}.\, \mathrm{b}\,\langle f\, \mathrm{e}, F\,(\ell x^{o}.\, \mathrm{c}\,(f\,(\mathrm{d}\, x)))\rangle)(\ell y^{o}.\, \mathrm{c}\,(\mathrm{d}\, y)) : o$$

$$\frac{}{\Gamma, x :: \varphi \mid \Delta \vdash_{\mathrm{ap}} x :: \varphi} \qquad \frac{}{\Gamma \mid \Delta, x :: \&_{i\in I}\varphi_i \vdash_{\mathrm{ap}} \pi_i\, x :: \varphi_i} \qquad \frac{\Gamma \mid \Delta \vdash_{\mathrm{ap}} t_i :: \varphi_i \quad (i \in I)}{\Gamma \mid \Delta \vdash_{\mathrm{ap}} \langle t_i \mid i \in I\rangle :: \&_{i\in I}\varphi_i}$$

$$\frac{\Gamma \mid \Delta_1 \vdash_{\mathrm{ap}} t :: \varpi \multimap \varphi \qquad \Gamma \mid \Delta_2 \vdash_{\mathrm{ap}} u :: \varpi}{\Gamma \mid \Delta_1, \Delta_2 \vdash_{\mathrm{ap}} t\,u :: \varphi} \qquad \frac{\Gamma \mid \Delta \vdash_{\mathrm{ap}} t :: \varphi_1 \to \varphi_2 \qquad \Gamma \mid _ \vdash_{\mathrm{ap}} u :: \varphi_1}{\Gamma \mid \Delta \vdash_{\mathrm{ap}} t\,u :: \varphi_2}$$

Fig. 3. Applicative terms



All subterms of t have purely linear kinds, so its *linear order* is 0. The infinite tree $\mathrm{BT}(t)$ generated starts as pictured on the left. The maximal branches of this infinite tree have the form $\mathrm{b}^n\mathrm{c}^n\mathrm{d}^n\mathrm{e}$ for all $n \in \mathbb{N}$, which is not context-free. Thus, due to the correspondence between order-1 higher-order recursion schemes and pushdown automata (e.g. [Hague et al. 2008]), $\mathrm{BT}(t)$ cannot belong to BT_1. However, it does belong to BT_2: it is generated by the order-2 HORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$, where $\Sigma = \{\mathrm{b} :: o \to o \to o, \mathrm{c} :: o \to o, \mathrm{d} :: o \to o, \mathrm{e} :: o\}$, $\mathcal{N} = \{S :: o, F :: (o \to o) \to o, G :: o \to o, H :: (o \to o) \to o \to o\}$, and $\mathcal{R}$ contains the following rules:

$$S = FG \qquad Ff = b(f\mathrm{e})(F(Hf)) \qquad Gx = \mathrm{c}(\mathrm{d}x) \qquad Hfx = \mathrm{c}(f(\mathrm{d}x))$$

Remark 12. As shown above, for schemes, a fixed linear order may turn out to be more expressive than the corresponding standard order. This extra expressivity will turn out to have surprisingly little cost in terms of complexity: we will still be able to show that (under certain constraints) the corresponding model-checking problem is in n-EXPTIME, where n is the linear order. However, there will be a price to pay in terms of the kind of properties that can be verified, as captured by the forthcoming definition of automata (Definition 21).

3 AN ALTERNATIVE: LINEAR HIGHER-ORDER RECURSION SCHEMES

Higher-order model-checking can be expressed in terms of the λY-calculus, or in terms of higher-order recursion schemes [Salvati and Walukiewicz 2016]. The new framework described earlier was introduced as a generalization of the λY-calculus. Now, we show that it can also be presented as a generalization of HORS, which we call *linear higher-order recursion schemes (LHORS)*.

3.1 Definition of LHORS

Though based on terms of $\lambda\!\!\lambda Y$, LHORS are superficially different: rather than having fixpoint operators, they handle recursion through a list of mutually recursive function definitions. Whereas the $\lambda\!\!\lambda Y$-calculus comes from the tradition of the λ-calculus, LHORS (like HORS) are inspired by grammars [Damm 1977] and program schemes [Nivat 1972].

3.1.1 Definition. A term $t \in \mathsf{KT}_{\Gamma|\Delta}(\kappa)$ is called **applicative** if one can derive $\Gamma \mid \Delta \vdash_{\mathrm{ap}} t :: \kappa$ using the kinding rules from Figure 3. An applicative term is necessarily $\triangleright_{\beta\eta\delta}$-normal. It does not contain any fixpoints or abstractions, and only consists of pairing and applying (projections of) variables from the contexts. We write $\mathsf{App}_{\Gamma|\Delta}(\kappa)$ for the set of applicative terms t such that $\Gamma \mid \Delta \vdash_{\mathrm{ap}} t :: \kappa$. A linear HORS will associate to every non-terminal a term of the form

$$t = \mathrm{l}_1 x_1. \ldots \mathrm{l}_n x_n. t' \in \mathsf{KT}_{\Gamma|_}(\varphi)$$

where $t' \in \mathsf{App}_{\Gamma, V_\lambda | V_\ell}(o)$, $\mathrm{l}_i \in \{\lambda, \ell\}$ and $V_{\mathrm{l}} = \{x_i \mid \mathrm{l}_i = \mathrm{l}\}$. We call such terms **function definitions of kind φ in context** Γ, and write $\mathsf{Def}_\Gamma(\varphi)$ for the corresponding set.

Definition 13. A **linear HORS (LHORS)** is a 4-tuple $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$ where:

- Σ is a tree signature,
- $\mathcal{N}$ is a finite set of kinded **non-terminals**, with a functional kind; we use upper-case letters $F, G, H, \dots$ to range over them. We denote $\mathcal{N}(F)$ the functional kind of F and write $F :: \mathcal{N}(F)$.
- $S \in \mathcal{N}$ is a distinguished **start symbol** of kind o,
- $\mathcal{R}$ is a function associating to each F in $\mathcal{N}$ a kinded term $\mathcal{R}(F) \in \mathrm{Def}_{\Sigma,\mathcal{N}}(\mathcal{N}(F))$.

We define some measures on LHORS: the **linear order** (*resp.* **linear depth**) of a LHORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$, written $\ell o(\mathcal{G})$ (*resp.* $\ell d(\mathcal{G})$), is the maximal linear order (*resp.* linear depth) of the kinds of its non-terminals in $\mathcal{N}$. Its **size** is

$$|\mathcal{G}| = \sum_{F:\varphi\in\mathcal{N}} |\mathcal{N}(F)|.$$

Example 14. The HORS of Figure 11 can be given linear kinds and presented as a LHORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$ where Σ is the tree signature given in Example 11, the non-terminals $\mathcal{N} = \{S :: o, F :: (o \multimap o) \multimap o, G :: o \multimap o, H :: (o \multimap o) \multimap o \multimap o\}$ are kinded simply by replacing the non-linear arrows with linear arrows in the HORS of Example 11; and, finally:

$$\begin{array}{rcllcl}
\mathcal{R}(S) & = & F\,G & :: & o \\
\mathcal{R}(F) & = & \ell f^{o\multimap o}.\, b\,\langle f\,\mathrm{e}, F\,(H\,f)\rangle & :: & (o \multimap o) \multimap o \\
\mathcal{R}(G) & = & \ell x^{o}.\, \mathrm{c}\,(\mathrm{d}\,x) & :: & o \multimap o \\
\mathcal{R}(H) & = & \ell f^{o\multimap o}.\, \ell x^{o}.\, \mathrm{c}\,(f\,(\mathrm{d}\,x)) & :: & (o \multimap o) \multimap o \multimap o
\end{array}$$

3.1.2 Value Tree of a LHORS. Given a LHORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S\rangle$ we now define the infinite tree it generates: its *value tree*. It is possible to do so using an ω-CPO of trees, or using Böhm evaluation, but we favour here an alternative presentation which underlies our proof of soundness and completeness of the intersection type system to come. It is a variant of a relation introduced by Kobayashi and Ong [2009] for proving soundness and completeness of their system. The value tree is computed by iterating a rewriting relation $\triangleright$ on applicative terms.

We begin by defining **tree contexts**:

$$T[-] ::= [-] \mid \mathrm{a}\, t_1 \,\dots\, t_{i-1}\, T[-]\, t_{i+1} \,\dots\, t_n \mid \langle t_1, \dots, t_{i-1}, T[-], t_{i+1}, \dots, t_n\rangle$$

where a is a terminal symbol in Σ. We give a reduction on applicative terms $t \in \mathrm{App}_{\Sigma,\mathcal{N}|_}(o)$ by:

$$\begin{array}{rcll}
T[F\, t_1 \,\dots\, t_n] & \triangleright & T[t[t_i/x_i]] & (\mathcal{R}(F) = l_1 x_1. \dots l_n x_n.\, t) \\
T[(\pi_j\, \langle t_i \mid i \in I\rangle)\, u_1 \,\dots\, u_p] & \triangleright & T[t_j\, u_1 \,\dots\, u_p] &
\end{array}$$

Definition 15. A linear HORS $\mathcal{G}$ is *productive* when the limit of any potentially infinite sequence of reductions $S \triangleright \mathcal{R}(S) \triangleright t_2 \triangleright \cdots$ which is *fair*, *i.e.* which eventually rewrites everything that can be rewritten, exists. This limit is then called the **value tree** $\mathcal{V}(\mathcal{G})$ of $\mathcal{G}$.

On productivity. In the traditional definition of the value tree of a higher-order recursion scheme as the limit of a sequence of trees, see e.g. [Kobayashi and Ong 2009], the value tree always exists, but may contain a divergence symbol Ω, in which case the HORS is said to be *unproductive*. Various techniques to detect and enforce productivity exist, one of them using an interpretation of the HORS in an appropriate model [Haddad 2013]. Another possibility, which we adapt here from [Serre 2013, p.61], is to introduce a new unary symbol $\varepsilon : \&_{\{\star\}}\, o \multimap o$ in the tree signature Σ, and to replace every reduction rule $\mathcal{R}(F) = \mathrm{l}x_1 \dots \mathrm{l}x_n.\, t$ with $\mathcal{R}(F) = \mathrm{l}x_1 \dots \mathrm{l}x_n.\, \varepsilon\, t$. This can also be applied to a linear HORS $\mathcal{G}$, and always results in a *productive* linear HORS $\mathcal{G}^{\varepsilon}$.

To do model-checking, it then remains to lift the property we want to specify over the tree $\mathcal{V}(\mathcal{G})$ and checked by an APTA $\mathcal{A}$ to a property checked by an extended APTA $\mathcal{A}^{\varepsilon}$ over $\mathcal{V}(\mathcal{G}^{\varepsilon})$. Several choices are possible. One can, for instance, choose to reject all branches of the form $a_1 \cdots a_n \cdot \varepsilon^{\omega}$ (where $a_n \neq \varepsilon$), which correspond to branches ending in diverging computation. Dually, one may

wish to accept them. We shall take a more liberal approach and let the specifier choose whether a diverging branch $a_1 \cdots a_n \cdot \varepsilon^\omega$ should be accepted or not depending on the state in which the APTA is when it visits a_n. Each of these options can be implemented by building $\mathcal{A}^\varepsilon$ from $\mathcal{A}$ through simple modifications, such as additions of new states.

In what follows, we therefore suppose that all linear HORS are productive, replacing $\mathcal{G}$ with $\mathcal{G}^\varepsilon$ if needed. This will allow us to build on [Kobayashi and Ong 2009], in which (standard) HORS are all supposed to be productive.

3.2 Equivalence of $\lambda\!\!\lambda\mathbb{Y}$ and LHORS

In the rest of this section, we show that there are polynomial-time transformations between $\lambda\!\!\lambda\mathbb{Y}$ and LHORS, preserving the linear order and the generated infinite tree.

3.2.1 From LHORS to $\lambda\!\!\lambda\mathbb{Y}$. Consider a LHORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, and write $\mathcal{N} = F_1 :: \varphi_1, \ldots, F_n :: \varphi_n$, with $S = F_{i_0}$. We are going to emulate the recursive definitions via a single use of the fixpoint, exploiting product kinds in $\lambda\!\!\lambda\mathbb{Y}$. We define a term $\Sigma \mid _ \vdash \mathcal{R}^* : \&_{1\leq i\leq n}\varphi_i$ by:

$$\mathcal{R}^* = \mathbb{Y} R^{\&_{1\leq i\leq n}\varphi_i}.\, \langle \mathcal{R}(F_i)[\pi_i\, R/F_i] \mid i \in \{1, \cdots, n\} \rangle$$

Proposition 16. *For any LHORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, there is $\Sigma \mid _ \vdash \mathcal{G}^\circledast : o$ defined as*

$$\mathcal{G}^\circledast = \mathcal{R}(S)[\pi_i\, (\mathcal{R}^*)/F_i]$$

satisfying:

$$\mathrm{BT}(\mathcal{G}^\circledast) = \mathcal{V}(\mathcal{G}) \qquad \ell o(\mathcal{G}^\circledast) = \ell o(\mathcal{G}) \qquad \ell d(\mathcal{G}^\circledast) \leq \ell d(\mathcal{G}) + 1 \qquad |\mathcal{G}^\circledast| = O(poly(|\mathcal{G}|))$$

Proof. The first equality follows *e.g.* by co-induction, as the chains of reductions are essentially the same (with a few additional β-reduction steps for $\mathcal{G}^\circledast$). The second and third conditions can be verified directly. For the last one, we have $|\mathcal{G}^\circledast| = O(|\mathcal{G}|^2)$ because of the substitution. □

This translation from LHORS to $\lambda\!\!\lambda\mathbb{Y}$ is particularly simple, because the shape of the LHORS recursive definition along with its dynamics can be directly replicated in $\lambda\!\!\lambda\mathbb{Y}$ thanks to products. In contrast, the correspondence between the λY-calculus and standard HORS is much less direct. In the absence of products, Salvati and Walukiewicz [2016] show how to emulate a HORS by a sequence of fixpoint definitions. However, the approach involves a sequence of substitutions that may cause an explonential blow-up, meaning that the outcome need not be of polynomial size.

3.2.2 From $\lambda\!\!\lambda\mathbb{Y}$ to LHORS. The translation from $\lambda\!\!\lambda\mathbb{Y}$ to LHORS will proceed along the lines of [Salvati and Walukiewicz 2016], with complications due to linear variables and products. Another difference is that *op. cit.* performs β-reduction first, in order to focus on translating β-normal terms. In contrast, we do not rely on prior β-reduction, so as to obtain a polynomial-time transformation.

Lemma 17. *For any $\Sigma, \Gamma \mid \Delta \vdash t :: \kappa$, there exist $t^\odot, \mathcal{N}, \mathcal{F}$ such that $\mathcal{N} = \{F_1 :: \varphi_1, \ldots, F_n :: \varphi_n\}$ is a set of kinded non-terminals, $t^\odot$ is an applicative term satisfying $\Sigma, \Gamma, \mathcal{N} \mid \Delta \vdash_{\mathrm{ap}} t^\odot :: \varphi$, and $\mathcal{F}$ associates to each F_i some $\mathcal{F}(F_i) \in \mathrm{App}_{\Sigma, \mathcal{N}}(\varphi_i)$. Moreover, we have*

$$\begin{aligned}
\mathrm{BT}(t^\odot[\pi_i\, (\mathbb{Y}\mathcal{F})/F_i]) &= \mathrm{BT}(t) \\
\max(\ell o(\Gamma) + 1, \ell o(\Delta), \ell o(t^\odot), \max_{i\in I} \ell o(\varphi_i)) &= \max(\ell o(\Gamma) + 1, \ell o(\Delta), \ell o(t)) \\
\max(\ell d(t^\odot), \max_{F_i\in\mathcal{N}} \ell d(\varphi_i)) &= \ell d(t) \\
|t^\odot| + \sum_{1\leq i\leq n} |\mathcal{F}(F_i)| &= O(poly(|t|))
\end{aligned}$$

Proof. For the transformation to LHORS, it suffices to have the above transformation for terms $\Sigma \mid _ \vdash t :: o$, but the more general statement given permits a direct proof by induction on t.

Given $\Sigma \mid _ \vdash t :: o$, for every subterm u of t of the form $\lambda x^{\varphi}. u'$, $\ell x^{\kappa}. u'$ or $\mathbb{Y} x^{\kappa}. u'$, we are going to introduce a non-terminal F_u in $\mathcal{N}$, so as to replace u with F_u. However, we cannot quite do that, because u' is not *closed* (not counting variables in Σ). Hence, we perform λ-lifting on the fly. □

As a consequence, we obtain:

Proposition 18. *For any $\Sigma \mid _ \vdash t :: o$ there is a LHORS $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ such that*

$$\mathcal{V}(\mathcal{G}) = \mathrm{BT}(t) \qquad \ell\mathrm{o}(\mathcal{G}) = \ell\mathrm{o}(t) \qquad \ell\mathrm{d}(\mathcal{G}) = \ell\mathrm{d}(t) \qquad |\mathcal{G}| = O(poly(|t|)).$$

Proof. By Lemma 17, there are $(t^{\odot}, \mathcal{N}, \mathcal{F})$ as above (write $\mathcal{N} = F_1 :: \varphi_1, \ldots, F_n :: \varphi_n$). Set $\mathcal{N}' = \mathcal{N}, F_{n+1} :: o$; and $\mathcal{F}' = \mathcal{F} \uplus \{F_{n+1} \mapsto t^{\odot}\}$. We almost got a linear recursion scheme, except that the $\mathcal{F}(F_i)$ are not necessarily η-expanded. We can perform η-expansion in polynomial time, yielding a LHORS $\langle \Sigma, \mathcal{N}', \mathcal{F}', F_{n+1} \rangle$ such that

$$\mathrm{BT}(t) = \mathrm{BT}(\mathcal{R}(S)[\pi_i\,(\mathbb{Y}\,\mathcal{F}')/F_i])$$

Hence we conclude that $\mathrm{BT}(t) = \mathcal{V}(\mathcal{G})$ by Proposition 16. □

This completes the proof of the equivalence between $\lambda\!\ell Y$ and LHORS. Not only have we proved that, for any linear order n, the $\lambda\!\ell Y$-terms and the LHORS of linear order n generate the same trees, but also we have established that the translations in both directions preserve linear depth and cause only a polynomial increase in size. We stress that they can be computed in polynomial time. Therefore, $\lambda\!\ell Y$ and LHORS are completely interchangeable from the point of view of the higher-order model-checking problem. When developing the model-checking algorithm via intersection type in the next section, we will work with LHORS. As their structure is more constrained, they provide a more comfortable setting for our analysis. On the other hand, when *using* the result, we will appeal to the more liberal $\lambda\!\ell Y$-calculus as a target for translations, in Sections 5 and 6.

4 AN INTERSECTION TYPE SYSTEM FOR MODEL-CHECKING LINEAR HORS

4.1 Linear-Nonlinear Alternating Parity Tree Automata

From alternating parity tree automata to intersection types. Higher-order model-checking has traditionally considered the verification of MSO (equivalently, modal μ-calculus) formulæ over the infinite trees generated by HORS. The problem is typically solved by translating the formula of interest into an equivalent alternating parity tree automaton (APTA), and then lifting the behaviour of this APTA from trees to HORS. Below we recall the connection between APTA and intersection types, which was first made in [Kobayashi 2009] for Büchi automata and then extended to APTA in [Kobayashi and Ong 2009]. A typical transition of an APTA over a binary symbol $\mathrm{a} :: o \to o \to o$ is $\delta(q, \mathrm{a}) = [(1, q_0)] \vee [(2, q_1) \wedge (2, q_2)]$, meaning that when the APTA encounters a in state q, it can choose nondeterministically between the following:

- $(1, q_0)$, which visits only the first subtree (the left child of a) in state q_0, and drops the other subtree so that the run-tree will have only *one* child at the current position;
- $(2, q_1) \wedge (2, q_2)$, which drops the first subtree and visits the second one *twice*, duplicating it in the run-tree (starting an execution from state q_1 in one copy and from state q_2 in the other).

This behaviour allowing duplication or erasure during the run of an automaton is called *alternation*. Its connections with linear logic have been studied in [Grellois and Melliès 2015b]. In addition, APTAs use a *parity* condition: a function Ω attributes a *colour* $\Omega(q) \in \mathbb{N}$ to each state $q \in Q$. Then, a run-tree is *accepting* if and only if the maximal colour seen infinitely often along each branch is even. When there is an accepting run-tree, the tree over which the APTA runs satisfies the MSO

property checked by the automaton. Kobayashi and Ong observed that the transition mentioned above amounts to typing the symbol a with the intersection types

$$\mathrm{a} : \bigwedge_{\{\star\}} \Box_{\Omega(q_0)}\, q_0 \to \bigwedge \emptyset \to q \qquad \text{and} \qquad \mathrm{a} : \bigwedge \emptyset \to \bigwedge_{i \in \{1,2\}} \Box_{\Omega(q_i)}\, q_i \to q$$

in a type system that we are about to define (our precise formulation is informed by later work [Grellois and Melliès 2015b]). In this way, they gave a correspondence between APTAs and sets of intersection types for terminals. Accordingly, every (accepting) run-tree of an APTA over a tree t can be translated to a (winning) typing derivation in the intersection type system, see for instance [Grellois 2016]. In what follows, exploiting this type-theoretic intuition, we redefine APTA and define LNAPTA as sets of typings rather than usual transition-based definitions.

Intersection types refining kinds. In [Kobayashi and Ong 2009], the intersection types that describe APTA behaviour over symbol a are required to *refine* the kinding $\Sigma(\mathrm{a})$ of the tree constructor a. We extend the approach, taking into account the fact that our system of kinds is more elaborate (we can form products and take linear arrows). This results in a richer set of intersection types.

Definition 19 (Intersection types). Let Q be a finite set of *states* and *Col* a finite set of *colours*. The **intersection types** σ over Q and *Col* are defined by the grammar below.

$$\begin{array}{rcll} \sigma & ::= & q \mid P \multimap \sigma \mid E \multimap \sigma \mid A \to \sigma & (q \in Q) \\ A & ::= & \bigwedge_{i \in I} \Box_{c_i}\, \sigma_i & (c_i \in Col) \\ P & ::= & \langle\, \emptyset, \ldots, \emptyset, \Box_c\, \sigma, \emptyset, \ldots, \emptyset \,\rangle & (c \in Col) \\ E & ::= & \langle\, \emptyset, \ldots, \emptyset, \ldots, \emptyset \,\rangle & \end{array}$$

We say that a type is *product-free* if it is generated by the grammar $\sigma \ ::= \ q \mid A \to \sigma$ $(q \in Q)$. The intersection connective is assumed to be associative, commutative and idempotent. We suppose also that it is stable under reindexing, and we identify intersection types modulo reindexing.

A refinement relation ensures that types formed by intersection share a common structure:

Definition 20 (Refinement). The **refinement relation** between intersection types and kinds is defined by the following rules.

$$\frac{q \in Q}{q :: o} \qquad \frac{P :: \varpi \qquad \sigma :: \varphi}{P \multimap \sigma :: \varpi \multimap \varphi} \qquad \frac{E :: \varpi \qquad \sigma :: \varphi}{E \multimap \sigma :: \varpi \multimap \varphi} \qquad \frac{A :: \varphi \qquad \sigma :: \varphi'}{A \to \sigma :: \varphi \to \varphi'}$$

$$\frac{\forall i \in I,\ \sigma_i :: \varphi}{\bigwedge_{i \in I} \Box_{c_i}\, \sigma_i :: \varphi} \qquad \frac{\sigma :: \varphi_j}{\left\langle \emptyset, \ldots, \emptyset, \underbrace{\Box_c\, \sigma}_{\text{position } j}, \emptyset, \ldots, \emptyset \right\rangle :: \&_{i \in I} \varphi_i} \qquad \frac{}{E :: \varpi}$$

We may now define the new notion of linear-nonlinear APTA in a way reminiscent of the connection between APTA and intersection types discussed above.

Definition 21 (Linear-nonlinear APTA). A *linear-nonlinear APTA* (LNAPTA) is a tuple $\langle \Sigma, Q, \delta, q_0 \rangle$, where Σ is a tree signature, Q is a finite set of states, $q_0 \in Q$ is the inital state, and δ is a map from Σ to sets of intersection types over Q and *Col* such that $\sigma :: \Sigma(\mathrm{a})$ for any $\mathrm{a} \in \Sigma$ and $\sigma \in \delta(\mathrm{a})$.

The usual notion of APTA (without colouring) can now be recovered as follows.

Definition 22 (APTA). A linear-nonlinear APTA $\langle \Sigma, Q, \delta, q_0 \rangle$ will be called an APTA if, for $\mathrm{a} \in \Sigma$ and $\sigma \in \delta(\mathrm{a})$, the type σ is product-free.

Suppose there is $\Omega : Q \to Col$ such that, for all $a \in \Sigma$, each $\sigma \in \delta(a)$ has the shape

$$\bigwedge_{i_1 \in I_1} \square_{c_{i_1}} q_{i_1} \to \cdots \to \bigwedge_{i_n \in I_n} \square_{c_{i_n}} q_{i_n} \to q$$

where $c_{i_j} = \Omega(q_{i_j})$ for all $1 \leq j \leq n$ and $i_j \in I_j$. Then this type-theoretic definition is precisely equivalent to the usual notion of APTA, as defined, for instance, in [Kobayashi and Ong 2009]. In the remainder of the paper, we shall assume that APTA always come with such Ω so that we can write $\langle \Sigma, Q, \delta, q_0, \Omega \rangle$. Similarly, we shall only consider LNAPTA whose delinearizations into APTA (to be introduced) satisfy this assumption.

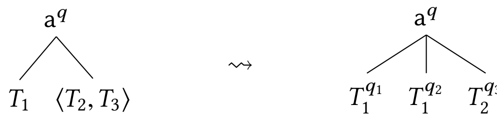
It should be noted that, on signatures whose kindings have particular shapes, LNAPTA correspond to various known classes of APTA. It has already been remarked by Melliès [Grellois and Melliès 2015b; Melliès 2014] that APTA restricted according to linearly kinded signatures would correspond to non-deterministic parity tree automata. In the more general framework of LNAPTA, this observation amounts to saying that LNAPTA over signatures with kinds of the shape $\&_{\{\star\}}\, o \multimap \cdots \multimap \&_{\{\star\}}\, o \multimap o$ yield, via delinearization, non-deterministic parity tree automata. A similar result can be given for the class of *disjunctive* APTA [Kobayashi and Ong 2011]: they are precisely the LNAPTA over signatures whose kinds are all of the form $\&_{i \in I}\, o \multimap o$. To our knowledge, this latter observation is new.

Example 23. Consider the tree signature $\Sigma = \{\mathrm{b} :: o \,\&\, o \multimap o, \mathrm{c} :: o \multimap o, \mathrm{d} :: o \multimap o, \mathrm{e} :: o\}$ of Example 11. Given a LHORS over this signature, suppose we want to check that, on any branch, after a c is encountered, we never encounter b again and we eventually encounter e (call this property P). By the remark above, any LNAPTA over Σ is disjunctive and, hence, cannot express P, which is not a disjunctive property [Kobayashi and Ong 2011]. However, we can express the disjunctive property that some branch satisfies $\neg P$ by the LNAPTA $\mathcal{A} = \langle \Sigma, \{q_0, q_1\}, \delta, q_0 \rangle$, where:

$$\begin{array}{rcl} \delta(\mathrm{b}) & = & \{\langle \square_1 q_0, \emptyset \rangle \multimap q_0, \langle \emptyset, \square_1 q_0 \rangle \multimap q_0, \langle \emptyset, \emptyset \rangle \multimap q_1\} \\ \delta(\mathrm{c}) & = & \{\langle \square_2 q_1 \rangle \multimap q_0, \langle \square_2 q_1 \rangle \multimap q_1\} \\ \delta(\mathrm{d}) & = & \{\langle \square_1 q_0 \rangle \multimap q_0, \langle \square_2 q_1 \rangle \multimap q_1\} \end{array}$$

The reader may convince themselves that this LNAPTA expresses the correct property by considering the corresponding disjunctive APTA. Following [Kobayashi and Ong 2011], such a disjunctive APTA expressing $\neg P$ on branches may be obtained systematically by complementing a deterministic parity word automaton for P and reformulating it as a (disjunctive) APTA running on branches. Using our model-checking algorithm to come, we may then check that a LHORS satisfies P by checking that it is rejected by $\mathcal{A}$.

LNAPTA run-tree. A LNAPTA can be seen as an APTA in the traditional sense, but defined in a more structured way and running over trees containing tuples. In fact, we shall define LNAPTA run-trees using the standard notion of run-trees for a traditional APTA. The idea is that a LNAPTA running over the tree a $T_1\ \langle T_2, T_3 \rangle$ is *constrained* with respect to a usual APTA: it can pick a transition nondeterministically but then, while it can duplicate or drop T_1, it has to run over *exactly one* copy of either T_2 or T_3. For instance, the type $\mathrm{a} : \bigwedge_{i \in \{1,2\}} \square_{\Omega(q_i)}\, q_i \to \langle \square_{\Omega(q_3)}\, q_3, \emptyset \rangle \multimap q$ corresponds to a transition whose effect on the production of a run-tree will be:



where the automaton duplicates T_1 and explores T_2 exactly once. Consequently, the resulting run-tree is a tree in the usual sense: there are no products, which correspond to choices that must be

resolved during the execution of a LNAPTA. Observe that the previous transition could have been performed by an APTA in the traditional sense, executed on the tree a $T_1\ T_2\ T_3$. This tree will be called the *delinearization* of a $T_1\ \langle T_2,\ T_3\rangle$ over which we shall execute the *delinearization* of the LNAPTA in question to extract a run-tree.

Definition 24 (Delinearizations).

- The delinearization $\{\!|\varphi|\!\}$ of a kind is defined as follows.

$$\{\!|o|\!\} \;=\; o \qquad \text{and} \qquad \{\!|\varphi_1 \to \varphi_2|\!\} \;=\; \{\!|\varphi_1|\!\} \to \{\!|\varphi_2|\!\}$$
$$\{\!|\&_{i \in I}\varphi_i \multimap \varphi|\!\} \;=\; \{\!|\varphi_1|\!\} \to \cdots \to \{\!|\varphi_n|\!\} \to \{\!|\varphi|\!\}$$

- The delinearization $\{\!|\Sigma|\!\}$ of a tree signature Σ maps a to $\{\!|\Sigma(\mathrm{a})|\!\}$
- The delinearization $\{\!|\sigma|\!\}$ (resp. $\{\!|A|\!\}$, $\{\!|P|\!\}$) of an intersection type is given below.

$$\begin{array}{rcl} \{\!|q|\!\} & = & q \\ \{\!|\langle\, \emptyset, \ldots, \emptyset, \Box_c\ \sigma, \emptyset, \ldots, \emptyset\,\rangle \multimap \tau|\!\} & = & \\ \multicolumn{3}{r}{\bigwedge\ \emptyset \to \cdots \to \bigwedge\ \emptyset \to \bigwedge_{\{\star\}}\ \Box_c\ \{\!|\sigma|\!\} \to \bigwedge\ \emptyset \to \cdots \to \bigwedge\ \emptyset\ \to\ \{\!|\tau|\!\}} \\ \{\!|\langle\, \emptyset, \ldots, \emptyset, \ldots, \emptyset\,\rangle \multimap \tau|\!\} & = & \bigwedge\ \emptyset \to \cdots \to \bigwedge\ \emptyset \to \cdots \to \bigwedge\ \emptyset\ \to\ \{\!|\tau|\!\} \\ \left\{\!\!\left|\bigwedge_{i\in I}\ \Box_{c_i}\ \sigma_i \to \tau\right|\!\!\right\} & = & \bigwedge_{i\in I}\ \Box_{c_i}\ \{\!|\sigma_i|\!\} \to \{\!|\tau|\!\} \end{array}$$

- The delinearization $\{\!|\mathcal{A}|\!\}$ of a linear-nonlinear APTA $\mathcal{A} = (\Sigma,\ Q,\ \delta,\ q_0)$ is the APTA $\{\!|\mathcal{A}|\!\} = (\{\!|\Sigma|\!\},\ Q,\ \{\!|\delta|\!\},\ q_0)$ such that $\{\!|\delta|\!\}(\mathrm{a}) = \{\{\!|\sigma|\!\} \mid \sigma \in \delta(\mathrm{a})\}$ for every $\mathrm{a} \in \Sigma$.
- Let Σ be a tree signature. The delinearization $\{\!|t|\!\}$ of a term $t \in \mathsf{KT}_{\Sigma|_}(o)$ is defined as

$$\left\{\!\!\left|\mathrm{a}\ \langle T_{11}, \ldots, T_{1k_1}\rangle\ \ldots\ \langle T_{n1}, \ldots, T_{nk_n}\rangle\right|\!\!\right\} \;=\; \mathrm{a}\ \{\!|T_{11}|\!\}\ \ldots\ \left\{\!\!\left|T_{1k_1}\right|\!\!\right\}\ \ldots\ \{\!|T_{n1}|\!\}\ \ldots\ \left\{\!\!\left|T_{nk_n}\right|\!\!\right\}$$

  where n can be 0 and where we used the convention that unary tuples of terms can be written instead of a term to uniformize notation.

We note the following consequences, which are easy to verify.

Lemma 25.

- *Suppose that $\tau :: \varphi$. Then $\{\!|\tau|\!\} :: \{\!|\varphi|\!\}$.*
- *Suppose that $t \in \mathsf{KT}_{\Sigma|_}(o)$. Then $\{\!|t|\!\} \in \mathsf{KT}_{\{\!|\Sigma|\!\}|_}(o)$.*

LNAPTA run-trees are now defined following the intuition that LNAPTAs are traditional APTAs obeying additional constraints on duplication and linearity in their executions.

Definition 26. Let $\mathcal{A}$ be a LNAPTA and $t \in \mathsf{KT}_{\Sigma|_}(o)$ be a ground term over the same signature Σ. A (accepting) run-tree of the delinearized APTA $\{\!|\mathcal{A}|\!\}$ over the delinearized term $\{\!|t|\!\}$ will be called a (accepting) *run-tree* of $\mathcal{A}$ over t.

4.2 The Intersection Type System $\mathcal{L}(\mathcal{A})$ for Linear-Nonlinear APTA Model-Checking

In this section, we define a generalization of the type system from [Kobayashi and Ong 2009] taking advantage of our finer notion of a kind. The intersection types we consider are the ones of Section 4.1. A *nonlinear context* Γ (resp. a *linear context* Δ) is a sequence of bindings

$$\Gamma \;=\; x_1\ :\ A_1\ ::\ \kappa_1,\ \ldots,\ x_n\ :\ A_n\ ::\ \kappa_n \qquad\qquad (\text{resp. } \Delta \;=\; y_1\ :\ P_1\ ::\ \kappa_1',\ \ldots,\ y_m\ :\ P_m\ ::\ \kappa_m')$$

where $A_i :: \kappa_i$ (resp. $P_j :: \kappa_j'$). We write $x : \Box_c\ \sigma \in \Gamma$ when the intersection type for x contains $\Box_c\ \sigma$. We also rely on two partial operations on contexts. The union $\Delta_1,\ \Delta_2$ of two linear contexts is defined if and only if they have disjoint domains, and contains the variables of Δ_1 and of Δ_2 with the unique intersection type and kind they have in Δ_1 and Δ_2. The intersection (or contraction) $\Gamma_1 \wedge \Gamma_2$ of nonlinear contexts contains:

- the variables of Γ_1 (resp. Γ_2) which are not in Γ_2 (resp. Γ_1) with the intersection type and kind they have in Γ_1 (resp. Γ_2),

$$\frac{\sigma \in \delta(\mathrm{a})}{_ \mid _ \vdash_{\mathcal{A}} \mathrm{a} : \sigma :: \Sigma(\mathrm{a})} \qquad \frac{x \notin \Sigma}{x : \bigwedge_{\{\star\}} \Box_{\varepsilon}\, \sigma :: \varphi \mid _ \vdash_{\mathcal{A}} x : \sigma :: \varphi}$$

$$\frac{x \notin \Sigma \cup \mathcal{N}}{_ \mid x : \langle \emptyset, \ldots, \emptyset, \Box_{\varepsilon}\, \sigma, \emptyset, \ldots, \emptyset \rangle :: \&_{i \in I} \varphi_i \vdash_{\mathcal{A}} \pi_i\, x : \sigma :: \varphi_i}$$

$$\frac{\Gamma, x : \bigwedge_{i \in I} \Box_{c_i}\, \sigma_i :: \varphi_1 \mid \Delta \vdash_{\mathcal{A}} t : \tau :: \varphi_2 \qquad I \subseteq J}{\Gamma \mid \Delta \vdash_{\mathcal{A}} \lambda x.\, t : \bigwedge_{j \in J} \Box_{c_j}\, \sigma_j \to \tau :: \varphi_1 \to \varphi_2} \qquad \frac{\Gamma \mid \Delta \vdash_{\mathcal{A}} t : \tau :: \varphi_2 \qquad x \notin \mathrm{dom}(\Gamma, \Delta)}{\Gamma \mid \Delta \vdash_{\mathcal{A}} \lambda x.\, t : \bigwedge \emptyset \to \tau :: \varphi_1 \to \varphi_2}$$

$$\frac{\Gamma \mid \Delta, x : P :: \varpi \vdash_{\mathcal{A}} t : \sigma :: \varphi}{\Gamma \mid \Delta \vdash_{\mathcal{A}} \ell x.\, t : P \multimap \sigma :: \varpi \multimap \varphi} \qquad \frac{\Gamma \mid \Delta \vdash_{\mathcal{A}} t : \sigma :: \varphi \qquad x \notin \mathrm{dom}(\Gamma, \Delta)}{\Gamma \mid \Delta \vdash_{\mathcal{A}} \ell x.\, t : E \multimap \sigma :: \varpi \multimap \varphi}$$

$$\frac{\Gamma_1 \mid \Delta_1 \vdash_{\mathcal{A}} t : \langle \emptyset, \ldots, \emptyset, \Box_c\, \sigma, \emptyset, \ldots, \emptyset \rangle \multimap \tau :: \varpi \multimap \varphi \qquad \Gamma_2 \mid \Delta_2 \vdash_{\mathcal{A}} u_j : \sigma :: \varpi}{\Gamma_1 \wedge \Box_c\, \Gamma_2 \mid \Delta_1, \Box_c\, \Delta_2 \vdash_{\mathcal{A}} t \,\langle u_1, \ldots, u_j, \ldots, u_n \rangle : \tau :: \varphi}$$

$$\frac{\Gamma \mid \Delta \vdash_{\mathcal{A}} t : E \multimap \sigma :: \varpi \multimap \varphi}{\Gamma \mid \Delta \vdash_{\mathcal{A}} t \,\langle u_1, \ldots, u_j, \ldots, u_n \rangle : \sigma :: \varphi} \qquad \frac{\Gamma \mid \Delta \vdash_{\mathcal{A}} u_j : \sigma :: \varphi \qquad j \in I}{\Gamma \mid \Delta \vdash_{\mathcal{A}} \pi_j \,\langle u_i \mid i \in I \rangle : \sigma :: \varphi}$$

$$\frac{\Gamma_0 \mid \Delta \vdash_{\mathcal{A}} t : \bigwedge_{i \in I} \Box_{c_i}\, \sigma_i \to \tau :: \varphi_1 \to \varphi_2 \qquad \forall i \in I,\ \ \Gamma_i \mid _ \vdash_{\mathcal{A}} u : \sigma_i :: \varphi_1}{\Gamma_0 \wedge \Box_{c_1}\Gamma_1 \wedge \cdots \wedge \Box_{c_n}\Gamma_n \mid \Delta \vdash_{\mathcal{A}} t\, u : \tau :: \varphi_2}$$

Fig. 4. The intersection type system $\mathcal{L}(\mathcal{A})$.

- and the variables x which appear in both Γ_1 and Γ_2 if they have the same kind κ in both; otherwise, the operation is undefined. We exploit stability of intersection types under reindexing to write $A_i = \bigwedge_{j \in J_i} \Box_{c_j}\, \sigma_j$ for the type of x in Γ_i, with $J_1 \cap J_2 = \emptyset$. Then x appears in $\Gamma_1 \wedge \Gamma_2$ as $x : \bigwedge_{j \in J_1 \uplus J_2} \Box_{c_j}\, \sigma_j :: \kappa$ (by idempotence some of these types may collapse).

We also define the operation $\Box_c$, for $c \in \mathit{Col}$, on types of the shape A, P and E and on contexts as follows:

$$\begin{array}{rcl} \Box_c\, \left(\bigwedge_{i \in I} \Box_{c_i}\, \sigma_i\right) & = & \bigwedge_{i \in I} \Box_{\max(c, c_i)}\, \sigma_i \\ \Box_c \,\langle \emptyset, \ldots, \emptyset, \Box_{c'}\, \sigma, \emptyset, \ldots, \emptyset \rangle & = & \left\langle \emptyset, \ldots, \emptyset, \Box_{\max(c, c')}\, \sigma, \emptyset, \ldots, \emptyset \right\rangle \\ \Box_c\, (x : A :: \varphi, \Gamma) & = & x : \Box_c\, A :: \varphi,\ \Box_c\, \Gamma \\ \Box_c\, (x : P :: \varphi, \Delta) & = & x : \Box_c\, P :: \varphi,\ \Box_c\, \Delta \end{array}$$

where $\Box_c$ applied to the empty context is again the empty context and $\Box_c\, E = E$.

Given a LNAPTA $\mathcal{A}$, the intersection type system is defined in Figure 4. Sequents are of the shape $\Gamma \mid \Delta \vdash_{\mathcal{A}} t : \sigma :: \varphi$, where we assume Γ and Δ to have disjoint domains.

As such, the type system is only a way to type the terms $\mathcal{R}(F)$, to which non-terminals of the given recursion scheme rewrite. To account for recursion, Kobayashi and Ong define a parity game that we extend to our setting. Note that, following [Grellois and Melliès 2015b], we introduce a *neutral* colour ε in the parity game, which is smaller than all others and losing.

Definition 27 (Typing game). Let $\mathcal{G}$ be a linear HORS and $\mathcal{A}$ be a linear-nonlinear APTA with an associated colouring function Ω. We define the typing game $Typ(\mathcal{G}, \mathcal{A})$ as the parity game $(V_\forall, V_\exists, (S, q_0, \varepsilon), E, \Omega)$, where:

- $V_\exists = \{(F, \sigma, c) \mid F \in \mathcal{N}, \sigma :: \mathcal{N}(F), c \in Col\}$
- $V_\forall = \{\Gamma \text{ s.t. } \mathrm{dom}(\Gamma) \subseteq \mathcal{N}\}$
- $E = \{((F, \sigma, c), \Gamma) \mid \Gamma \mid _ \vdash_{\mathcal{A}} \mathcal{R}(F) : \sigma :: \mathcal{N}(F)\} \uplus \{(\Gamma, (F, \sigma, c)) \mid F : \Box_c\, \sigma :: \mathcal{N}(F) \in \Gamma\}$
- The edges of the shape $(\Gamma, (F, \sigma, c))$ have colour c and the edges of the shape $((F, \sigma, c), \Gamma)$ have neutral colour ε.

The idea is that two players, Adam (who owns the vertices from $V_\forall$) and Eve (who owns those from $V_\exists$), build incrementally a typing as follows:

- Eve starts from (S, q_0, ε), and must answer with a context Γ such that $\Gamma \mid _ \vdash_{\mathcal{A}} \mathcal{R}(S) : q_0 :: o$. Γ contains typings for the nonterminals introduced when rewriting S to $\mathcal{R}(S)$.
- If Γ is empty, Eve wins. Otherwise, Adam picks a typed nonterminal $F : \Box_c \sigma :: \mathcal{N}(F) \in \Gamma$ and outputs the colour c.
- Then Eve provides a context Γ' such that $\Gamma' \mid _ \vdash_{\mathcal{A}} \mathcal{R}(F) : \sigma :: \mathcal{N}(F)$, and so on.

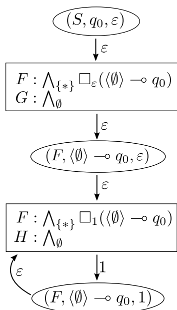
The interaction stops if Eve can answer with the empty context (she wins), if she cannot answer (she loses) or if the play is infinite (Eve wins iff the parity condition is satisfied). We can use the parity game to obtain a refined version of Kobayashi and Ong's soundness-and-completeness theorem.

Theorem 28 (Soundness and Completeness). *Let $\mathcal{G}$ be a linear HORS and $\mathcal{A}$ be a LNAPTA. Eve has a winning strategy in the typing game $Typ(\mathcal{G}, \mathcal{A})$ if and only if there is an accepting run-tree of $\mathcal{A}$ over the tree $\mathcal{V}(\mathcal{G})$ produced by $\mathcal{G}$.*

Proof. For Soundness, suppose that Eve has a winning strategy in $Typ(\mathcal{G}, \mathcal{A})$. One can define a series of notions of delinearization and, notably, one for proofs, which allows Eve to translate her moves in $Typ(\mathcal{G}, \mathcal{A})$ into moves of $Typ(\{\!|\mathcal{G}|\!\}, \{\!|\mathcal{A}|\!\})$, opening the same (up to delinearization of types) moves for Adam. It can be shown that Eve obtains a winning strategy in $Typ(\{\!|\mathcal{G}|\!\}, \{\!|\mathcal{A}|\!\})$ this way. Then we observe that the delinearized typing derivations that Eve uses are precisely typing derivations of the Kobayashi-Ong system, modulo the fact that sequents are written $\Gamma \mid _ \vdash_{\mathcal{A}} t : \sigma :: \kappa$ instead of $\Gamma \vdash t : \sigma :: \kappa$. It follows that the definition of our parity game $Typ(\{\!|\mathcal{G}|\!\}, \{\!|\mathcal{A}|\!\})$ coincides with their parity game for $\{\!|\mathcal{G}|\!\}$ and $\{\!|\mathcal{A}|\!\}$ (more precisely, it coincides with its modal rephrasal by Grellois and Melliès given in [Grellois 2016]). As a consequence of the original result of soundness of [Kobayashi and Ong 2009], there is an accepting run-tree of $\{\!|\mathcal{A}|\!\}$ over the tree $\mathcal{V}(\{\!|\mathcal{G}|\!\})$. By definition, there is an accepting run-tree of $\mathcal{A}$ over the tree $\mathcal{V}(\mathcal{G})$.

For Completeness, one can carefully adapt the argument from [Kobayashi and Ong 2009]. The main changes necessary concern product constructions and the presence of linear contexts. □

Example 29. Consider the LHORS $\mathcal{G}$ from Example 14 and the LNAPTA $\mathcal{A}$ from Example 23.



Despite the linear typing, $Typ(\mathcal{G}, \mathcal{A})$ is still too big to be presented fully. On the left, we display its small fragment, where circled nodes belong to Eve and rectangles to Adam.

This diagram follows a strategy for Eve in which she declares a typing that only explores the infinite branch on the right of the infinite tree from Example 14, encountering only occurrences of b; and a counter-strategy for Adam exploring only F. The infinite loop visits colour 1 infinitely often, therefore it is losing for Eve. In fact, Adam has a winning strategy in $Typ(\mathcal{G}, \mathcal{A})$: accordingly, $\mathcal{G}$ is rejected by $\mathcal{A}$, which corresponds to the fact that it satisfies the original property P considered in Example 23.

The parity game $Typ(\mathcal{G}, \mathcal{A})$ is finite; therefore, the LNAPTA model-checking problem for infinite trees generated by LHORS is decidable. Of course, that in itself is not new since a LHORS can be easily translated to a HORS generating the same infinite tree (but with higher order). The value of the development above will appear more clearly in the next section with the complexity analysis.

4.3 Complexity Analysis

We have seen that given a LHORS $\mathcal{G}$ and a LNAPTA $\mathcal{A}$, the model-checking problem is reduced to the existence of a winning strategy for Eve in the parity game $Typ(\mathcal{G}, \mathcal{A})$. Since the parity game is played on typing judgements of our intersection type system, the key element of the complexity analysis will be finding an upper bound on the number of intersection types refining a given kind. As pointed out before, linear kinds will affect the complexity in a relatively small manner.

4.3.1 Bounding the Size of Refinements. First, let us explain briefly why only non-linear arrows are expected to cause an exponential blow-up. Fix a LHORS $\mathcal{G}$ and a linear-nonlinear APT $\mathcal{A}$. The source of the tower of exponentials in the complexity of higher-order model-checking is the following refinement rule for non-linear arrows:

$$\bigwedge_{i \in I} \Box_{c_i}\, \sigma_i \to \sigma :: \varphi \to \psi$$

(which combines two refinement rules of Definition 20), where for all $i \in I$, $\sigma_i :: \varphi$ and $\sigma :: \psi$. The intersection types are idempotent, so assuming φ and ψ have only finitely many refinements it follows that the same holds for $\varphi \to \psi$. Writing $\sharp\kappa$ for the number of distinct intersection type refinements of κ, we get $\sharp(\varphi \to \psi) \leq 2^{C \sharp\varphi} \sharp\psi$, where C is the number of colours. It should be clear to the reader how non-linear arrows nested on the left will iterate this exponential, culminating in a tower of height equal to the order of the kind. The impact of the other kind constructors is milder:

$$\sharp(o) = Q \qquad \sharp(\&_{i \in I} \varphi_i) = \sum_{i \in I} C\, \sharp\varphi_i \qquad \sharp(\varpi \multimap \varphi) = (\sharp\varpi + 1)(\sharp\varphi)$$

where Q is the number of states of $\mathcal{A}$.

None of these cause directly an exponential blow-up. However, it would be naive to ignore their effect. From our discussions, one might expect that, for purely linear kinds κ, $\sharp\kappa$ will be linear in $|\kappa|$. Unfortunately, it is easy to see that this is not the case: the kind $o \multimap \ldots \multimap o \multimap o$ has size $O(n)$, but admits $O(Q^n)$ refinements. To circumvent the apparent issue, we introduce the notion of *linear depth*, capturing the maximal depth of isolated linear chunks of kinds.

In the purely non-linear case [Kobayashi and Ong 2009], the bound also involves the *maximal arity* of kinds. The presence of linear kinds forces us to switch to a generalized notion of arity.

Definition 30. We define the **local generalized arity** $\mathrm{ga}(\sigma)$ for a kind σ by:

$$\begin{array}{rclrcl} \mathrm{ga}(o) & = & 1 & \mathrm{ga}(\kappa \multimap \varphi) & = & \max(\mathrm{ga}(\kappa), \mathrm{ga}(\varphi)) + 1 \\ \mathrm{ga}(\&_{i \in I} \varphi_i) & = & \max_{i \in I} \mathrm{ga}(\varphi_i) & \mathrm{ga}(\varphi \to \psi) & = & 1 + \mathrm{ga}(\psi) \end{array}$$

We say that σ has **generalized arity** A iff for all subkind ι of σ, we have $\mathrm{ga}(\iota) \leq A$.

Assume from now on that kinds in $\mathcal{G}$ have linear depth D, and generalized arity A. For the key lemma below, we assume that the number of colours of $\mathcal{A}$ is bounded by Q (from the definition it is bounded by $Q + 1$ as the intersection type system uses an additional colour ε, but the requirement is easy to ensure with no loss of generality, for instance, by adding a dummy state) – we find it more convenient to give a bound expressed in Q than in C and Q, which can be proved by induction on κ.

Lemma 31. *For all $d \geq 0$, define $f_0(n, x, y, z) = (xy)^{2^n}$ and $f_{d+1}(n, x, y, z) = x^{yz2^n f_d(D, x, y, A)}$. Then, for all kinds κ appearing in $\mathcal{G}$, we have $\sharp\kappa \leq f_{\ell o(\kappa)}(\mathrm{lld}(\kappa), |\kappa| + 1, Q, \mathrm{ga}(\kappa))$.*

From there, it is relatively easy to derive the following by arithmetic reasoning:

Corollary 32. *If $\mathcal{G}$ has linear order n, linear depth D and generalized arity A, then for all κ in $\mathcal{G}$:* $\sharp\kappa \leq \exp_n(A\,2^{D+1}(Q|\kappa|)^{2^D+1})$, *where* $\exp_0(x) = x$, $\exp_{n+1}(x) = 2^{\exp_n(x)}$.

The tower is two steps higher than usual if one considers D as a variable. It is easy to come up with a sequence of purely linear kinds showing that the additional double exponential is unavoidable. However, if D is fixed, we get bounds of the form $\exp_n(O(poly(AQS)))$ for refinements of kinds of linear order n, generalized arity A, size S, with types built from Q states – as hoped. We will see later that this is not an unreasonable assumption, as our translations yield a small linear depth.

Finally, from the above one can deduce complexity bounds for solving $Typ(\mathcal{G}, \mathcal{A})$.

Proposition 33. *Given a LHORS $\mathcal{G}$ with N non-terminals, kinds of maximal size S, linear depth D and linear order n; and a linear-nonlinear APT $\mathcal{A}$ with colours bounded by p and states bounded by $Q \geq p$. For $n \geq 1$, the time complexity for solving $Typ(\mathcal{G}, \mathcal{A})$ is* $O(N^{\lceil p/2 \rceil + 2} \exp_n(O(2^D)(QS)^{O(2^D)}))$.

Proof. The following inequalities follow immediately from the definition of $Typ(\mathcal{G}, \mathcal{A})$ and Corollary 32 (using also that the generalized arity of a kind is always bounded by its size).

$$|V_\forall|, |V_\exists| \leq N \exp_n(O(2^D)(QS)^{O(2^D)}) \qquad |E| \leq N^2 \exp_n(O(2^D)(QS)^{O(2^D)})$$

The result follows by applying the algorithm from [Jurdziński 2000]. □

Finally, let us fix some $D \geq 1$. A LHORS is D-**deep** if its linear depth does not exceed D. Our main theorem then follows as a corollary of the above:

Theorem 34. *Assume $n \geq 1$. The time complexity of checking whether a LNAPTA $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, q_0 \rangle$ accepts the value tree of a D-deep LHORS $\mathcal{G}$ of linear order n is $\exp_n(O(poly(|\mathcal{Q}||\mathcal{G}|)))$. In particular, the problem is n-EXPTIME complete (hardness follows from [Ong 2006]).*

5 IMMEDIATE CONSEQUENCES

Although recent developments in higher-order program verification were prompted by a decidability result for HORS [Ong 2006], subsequent complexity results for richer formalisms did not appeal to the result directly. Instead, their authors were developing dedicated decision procedures.

There are at least two reasons for this. Firstly, HORS represent only the part of the control flow of a program coming from higher-order computation, while programs typically manipulate data, and their behaviour depends on the data. Secondly, programs typically follow a different evaluation strategy, such as call-by-value. Both problems can be dealt with using plain HORS: data types can be represented via their Church encoding, *e.g.* with $\mathsf{B}^* = o \to o \to o$, while call-by-value programs can be translated to HORS (call-by-name) with CPS. Unfortunately, both translations increase type order and suggest increases in complexity.

In contrast, we are going to show that, thanks to Theorem 34, LHORS (equivalently, the λY-calculus) are a suitable target for such translations. First, in this section, we shall derive optimal bounds for MSO model-checking in two extensions of HORS *by translation* into LHORS. In the next section, we shall follow the same methodology (translation into the λY-calculus) to handle accurately a resource verification problem in the call-by-value setting.

5.1 Recursion Schemes Over Finite Data Domains

Recursive schemes over finite data domains (RSFD) [Kobayashi et al. 2010] extend standard (non-linear) HORS with a new ground kind d representing a finite domain whose elements correspond to constants $d_1, \cdots, d_k$ of that kind. *Terms* of RSFDs are those of standard HORS extended with

constants $d_i :: \mathsf{d}$, $\mathsf{case}_o :: \mathsf{d} \to \underbrace{o \to \cdots \to o}_{k} \to o$ and the reduction rule $\mathsf{case}_o\, d_i\, u_1 \,\ldots\, u_k \rhd u_i$. Importantly, RSFD kinding rules force $u_1, \ldots, u_k$ to have the ground kind o. Terminals b are kinded as usual by $o \to \cdots \to o \to o$. For RSFDs, we prove:

Proposition 35. *Given a RSFD $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, we construct a $\lambda\!\mathcal{Y}$-term $\Sigma \mid _ \vdash \mathcal{G}^* :: o$ as follows. First, we define the translation on kinds and terms.*

$$
\begin{array}{rclrclrcl}
o^* & = & o & \mathsf{b}^* & = & \mathsf{b} & F^* & = & F \\
\mathsf{d}^* & = & \underbrace{o \multimap \cdots \multimap o}_{k} \multimap o & (t\,u)^* & = & t^*\,u^* & (\lambda x^\theta.\, t)^* & = & \lambda x^{\theta^*}.\, t^* \\
 & & & d_i^* & = & \ell x_1^o.\, \ldots\, \ell x_k^o.\, x_i & & & \\
(\varphi \to \psi)^* & = & \varphi^* \to \psi^* & \mathsf{case}_o^* & = & \lambda d^{\mathsf{d}^*}.\, \lambda x_1^o.\, \ldots\, \lambda x_k^o.\, d\, x_1 \,\ldots\, x_k & & &
\end{array}
$$

In this way, for each $\theta = \mathcal{N}(F)$, we obtain $\Sigma, \mathcal{N}^* \mid _ \vdash \mathcal{R}(F)^* :: \theta^*$. Next let us take a fixpoint over the product of the non-terminals as in Proposition 16. Then, $(-)^*$ sends an RSFD $\mathcal{G}$ of order n (note that $\mathrm{ord}(\mathsf{d}) = 0$) to a $\lambda\!\mathcal{Y}$-term $\mathcal{G}^*$ of linear order n, of size linear in that of $\mathcal{G}$, and linear depth k; producing the same tree.

This allows us to conclude that:

Corollary 36. *For a fixed finite domain d, the MSO model-checking problem on infinite trees generated by RSFDs of order n is n-EXPTIME complete.*

Proof. Hardness is obvious, since regular HORS are RSFDs. For the upper bounds, by Proposition 35 and Theorem 34, we get that LNAPTA model-checking is in n-EXPTIME. But the expressiveness of LNAPTA depends on non-terminals. Since they are kinded non-linearly, (as we observed in Section 4.1) they concide with regular APTAs in this case. Therefore, they can express arbitrary MSO properties. □

Note for completeness that Kobayashi et al. [2010] prove $(n-1)$-EXPTIME completeness for reachability, but do not address general MSO properties. Note also that the n-EXPTIME bounds only hold under the requirement that d is fixed, because the linear depth depends on the size of d – a constraint that will be lifted in the next subsection with a different translation.

5.2 Higher-Order Recursion Schemes with Cases

Higher-order recursion schemes with cases (HORSC) [Neatherway et al. 2012] are similar to RSFD, with a few significant distinctions. Firstly, branching is allowed on d as well as on o, *i.e.* we have case_o as well as $\mathsf{case}_{\mathsf{d}} :: \mathsf{d} \to \underbrace{\mathsf{d} \to \cdots \to \mathsf{d}}_{k} \to \mathsf{d}$. Furthermore, the elements $d_i \in \mathsf{d}$ are themselves terminal symbols, and can appear in the value tree. Of course, this means that terminal symbols can have kind d as well as o. In general, terminal symbols that are not in d have kinds of the form:

$$\mathsf{a} :: \beta_1 \to \cdots \to \beta_n \to o$$

where $\beta_i \in \{o, \mathsf{d}\}$; so in particular they have return kind o.

Translating HORSCs to LHORS/$\lambda\!\mathcal{Y}$ is slightly more elaborate than for RSFDs, though it remains rather simple. Given a HORSC $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, we first observe that Σ is many-sorted. In contrast, the translated LHORS will have terminal symbols in Σ^*, defined as having the same symbols as Σ but rekinded by replacing every occurrence of d with o. Then, we prove:

Proposition 37. *Given a HORSC $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ we construct a $\lambda\!\!\mathcal{Y}$-term $\Sigma^* \mid _ \vdash \mathcal{G}^* :: o$ as follows. First, we define the translation on kinds and terms.*

$$
\begin{array}{rcl@{\qquad}rcl}
o^* & = & o & d_i^* & = & \ell x^{\&_{1\le i\le k} o}.\,\pi_i\,x \\
\mathsf{d}^* & = & (\&_{1\le i\le k} o) \multimap o & \mathsf{a}^* & = & \lambda x_1^{\beta_1}\ldots x_n^{\beta_n}.\,\mathsf{a}\,(\mathrm{elim}_{\beta_i}\,x_i \mid 1\le i\le n) \\
(\varphi \to \psi)^* & = & \varphi^* \to \psi^* & x^* & = & x \qquad\qquad F^* = F \\
 & & & (t\,u)^* & = & t^*\,u^* \qquad (\lambda x^{\varphi}.\,t)^* = \lambda x^{\varphi^*}.\,t^* \\
(\mathrm{case}_o\,t\,u_1\,\ldots\,u_k)^* & = & t^*\,\langle u_i^* \mid 1\le i\le k\rangle & (u_i : o) & & \\
(\mathrm{case}_{\mathsf{d}}\,t\,u_1\,\ldots\,u_k)^* & = & \ell x^{\&_{1\le i\le k} o}.\,t^*\,\langle u_i^*\,x \mid 1\le i\le k\rangle & (u_i : \mathsf{d}) & &
\end{array}
$$

where $\mathrm{elim}_o\,x = x$, $\mathrm{elim}_{\mathsf{d}}\,x = x\,\langle d_i \mid 1 \le i \le k\rangle$*. Dealing with the recursive definition as before, this yields* $\Sigma^* \mid _ \vdash \mathcal{G}^* :: o$ *of linear order n, linear depth 2, size linear in that of $\mathcal{G}$; and the same value tree.*

Note the usage of product kinds, essential in order to represent the generalized case distinction of HORSC. As for RSFDs, from Proposition 37 and Theorem 34 we obtain:

Corollary 38. *The MSO model-checking problem on HORSC of order n is n-EXPTIME complete.*

Note that, unlike for our translation of RSFDs, the one for HORSC yields $\lambda\!\!\mathcal{Y}$-terms of constant linear depth thanks to the use of products. This eliminates the need for a fixed data domain assumption in the corollary above. A posteriori, of course, the same holds for RSFDs: an RSFD is in particular a HORSC.

5.3 Kinding of Terminals

As demonstrated above, Theorem 34 can often be used to obtain better worst-complexity bounds than those implied by the classic result [Ong 2006], provided we can introduce linearity into the underpinning types. This could also be done by typing *terminals* (i.e. tree nodes) through linear types, such as $o \multimap o \multimap o$ and $o\&o \multimap o$. Every regular HORS can be transformed into an equivalent LHORS (wrt the induced value tree) with terminals *rekinded* linearly, either *multiplicatively* (*i.e.* $o \multimap \ldots \multimap o \multimap o$) or *additively* (*i.e.* $o \,\&\, \cdots \,\&\, o \multimap o$).

As we have seen in Section 4.1, the kinding of terminals affects properties expressible with linear-nonlinear APTAs. In particular, if we adopt the additive linear kinding for terminals mentioned above, then LNAPTAs correspond to *disjunctive APTAs* [Kobayashi and Ong 2011], which admit $(n-1)$-EXPTIME model-checking on standard HORS of order n. Therefore, it is tempting to conjecture that rekinding the terminals as suggested above always reduces the linear order, saving one exponential. Unfortunately, that is not the case. It turns out that there exist HORS that (even with terminals kinded linearly) do not admit a refined linear kinding reducing the linear order. Thus, the linearity information conveyed by linear kinding in $\lambda\!\!\mathcal{Y}$ is too rough to reproduce the argument of [Kobayashi and Ong 2011], which exploits the flexibility of intersection types to reduce the search space. Still, terminals kinded with additive types will play an important role in the next section, where we tackle a less immediate application of our main result to call-by-value programs.

6 CALL-BY-VALUE PROGRAMS

To avoid the complexity blow-up due to CPS translation, Tsukada and Kobayashi [2014] gave a direct intersection type system to show that the reachability problem for depth-n call-by-value programs is n-EXPTIME-complete. We show that, using Theorem 34, one can recover their result through a linear CPS transformation, and the refined complexity analysis afforded by the Theorem does yield the optimal bound. Moreover, we show that the same complexity bound applies to the more general problem of resource verification [Kobayashi 2009] in the call-by-value setting.

$$\frac{}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{tt} : \mathsf{B}} \qquad \frac{}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{ff} : \mathsf{B}} \qquad \frac{}{\mathcal{E};\Gamma, x : A \vdash_{\mathcal{L}} x : A}$$

$$\frac{\mathcal{E}, f_i : E_i;\Gamma \vdash_{\mathcal{L}} u_j : A_j \qquad (1 \le j \le p_i)}{\mathcal{E}, f_i : E_i;\Gamma \vdash_{\mathcal{L}} f_i \langle u_j \mid 1 \le j \le p_i \rangle : B_i} \qquad \frac{\mathcal{E};\Gamma \vdash_{\mathcal{L}} t : \mathsf{B} \qquad \mathcal{E};\Gamma \vdash_{\mathcal{L}} u_1 : A \qquad \mathcal{E};\Gamma \vdash_{\mathcal{L}} u_2 : A}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{if}\, t\, u_1\, u_2 : A}$$

$$\frac{}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{skip} : \mathsf{U}} \qquad \frac{\mathcal{E};\Gamma \vdash_{\mathcal{L}} t : \mathsf{U} \qquad \mathcal{E};\Gamma \vdash_{\mathcal{L}} u : A}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} t;u : A} \qquad \frac{\mathcal{E};\Gamma \vdash_{\mathcal{L}} t : \mathcal{R}}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{use}_a\, t : \mathsf{U}}$$

$$\frac{\mathcal{E};\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash_{\mathcal{L}} t : B}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \lambda\langle x_1, \ldots, x_n\rangle.\, t : A_1 \times \cdots \times A_n \Rightarrow B} \qquad \frac{\mathcal{E};\Gamma, x : \mathcal{R} \vdash_{\mathcal{L}} t : A}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{new}\, x\, \mathtt{from}\, \mathtt{q}\, \mathtt{in}\, t : A}$$

$$\frac{\mathcal{E};\Gamma \vdash_{\mathcal{L}} t : A_1 \times \cdots \times A_n \Rightarrow B \qquad \mathcal{E};\Gamma \vdash_{\mathcal{L}} u_i : A_i}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} t\, \langle u_i \mid 1 \le i \le n \rangle : B} \qquad \frac{\mathcal{E};\Gamma \vdash_{\mathcal{L}} u_1 : A \qquad \mathcal{E};\Gamma \vdash_{\mathcal{L}} u_2 : A}{\mathcal{E};\Gamma \vdash_{\mathcal{L}} u_1 \oplus u_2 : A}$$

Fig. 5. Typing rules for $\mathcal{L}$

6.1 A Call-By-Value Language

We consider a language $\mathcal{L}$, extending the simply-typed call-by-value calculus with primitives for resource generation and access.

Definition 39 (Types). $\mathcal{L}$**-types** are defined by the grammar

$$A, B ::= \mathsf{B} \mid \mathsf{U} \mid \mathcal{R} \mid A_1 \times \cdots \times A_n \Rightarrow B$$

where B is the type of *booleans*, U is the *unit* type, and $\mathcal{R}$ is a type for *resources*, which could be used to model *e.g.* file, network or memory accesses. B and U (but not $\mathcal{R}$) will be referred to as *base* types.

Let us fix a set Act of **actions**. We write Act^∞ for the set of possibly infinite words on Act, *i.e.* it contains both finite and infinite words. We fix a deterministic parity automaton $\mathcal{A} = \langle Q, Act, \delta, q_0, \Sigma, F\rangle$, which describes the allowed ways to use the resource. Note that this automaton recognizes both finite and infinite words: a finite word w is recognized if the automaton reaches F after reading w, and it recognizes an infinite word according to the parity acceptance condition. We write $\mathsf{L}_\mathsf{q}(\mathcal{A})$ for the set of words (both finite and infinite) accepted by $\mathcal{A}$ from state q.

Definition 40 (Terms). $\mathcal{L}$**-terms** are defined by the grammar

$$\begin{array}{rcl} t, u, v & ::= & x \mid f \mid \lambda\langle x_1, \ldots, x_n\rangle.t \mid t\,\langle u_1, \ldots, u_n\rangle \mid \mathtt{tt} \mid \mathtt{ff} \mid \mathtt{if}\, t\, u_1\, u_2 \mid \\ & & \mathtt{skip} \mid t;u \mid t \oplus u \mid \mathtt{use}_a\, t \mid \mathtt{new}\, x\, \mathtt{from}\, \mathtt{q}\, \mathtt{in}\, u \end{array}$$

where $a \in Act$, $q \in Q$ and all the tuplings are non-empty.

$\mathcal{L}$-terms will be typed using typing judgements of the form $\mathcal{E};\Gamma \vdash_{\mathcal{L}} t : A$, where $\mathcal{E}$ is the *environment* comprising typed function names, which are to be defined later in the whole program, and Γ is a standard typing context. The typing rules appear in Figure 5.

Definition 41 (Definitions). A **function definition** in $\mathcal{L}$ is $\mathcal{D} ::= \{f_i = \lambda\langle x_{i,1}, \ldots, x_{i,p_i}\rangle.\, t_i\}_{i \le N}$, where for all $1 \le i \le N$ we have $\mathcal{E};_ \vdash_{\mathcal{L}} \lambda\langle x_{i,1}, \ldots, x_{i,p_i}\rangle.\, t_i : E_i$ where $E_i = A_{i,1} \times \cdots \times A_{i,p_i} \Rightarrow B_i$, B_i a *base type*, and $\mathcal{E} = \{f_i : E_i \mid 1 \le i \le N\}$. We will sometimes write $\mathcal{D}(f_i) = \lambda\langle x_{i,1}, \ldots, x_{i,p_i}\rangle.\, t_i$.

Definition 42 (Programs). A $\mathcal{L}$**-program** is a pair $(P, \mathcal{D})$, where $\mathcal{D}$ is a function definition and P is a term satisfying $\mathcal{E};_ \vdash_{\mathcal{L}} P : B$, where B is a base type.

$$
\begin{array}{rclrcl}
E[\mathsf{if}\ \mathtt{tt}\ u_1\ u_2] & \rightsquigarrow_v & E[u_1] & E[\mathsf{if}\ \mathtt{ff}\ u_1\ u_2] & \rightsquigarrow_v & E[u_2] \\
E[\mathsf{skip}; t] & \rightsquigarrow_v & E[t] & E[f_i\ \langle v_i \mid 1 \le j \le p_i \rangle] & \rightsquigarrow^{\mathcal{D}} & E[t_i[v_j/x_j]]
\end{array}
$$

$$E[(\lambda\langle x_i \mid i \in I\rangle.t)\ \langle v_i \mid i \in I\rangle] \rightsquigarrow_v E[t[v_i/x_i]]$$

Fig. 6. Resource-free reduction rules on programs in $\mathcal{L}$

$$
\begin{array}{rclrcll}
(E[t_1 \oplus t_2], n) & \xrightarrow{\tau} & (E[t_i], n) & (P, n) & \xrightarrow{\tau} & (Q, n) & (P \rightsquigarrow_v^{\mathcal{D}} Q) \\
(E[\mathsf{use}_a\ r], n) & \xrightarrow{a} & (E[\mathsf{skip}], n) & (E[\mathsf{use}_a\ r'], n) & \xrightarrow{\tau} & (E[\mathsf{skip}], n) & (r \neq r')
\end{array}
$$

$$
\begin{array}{rcll}
(E[\mathsf{new}\, x\, \mathsf{from}\, \mathsf{q}\, \mathsf{in}\, t], r) & \xrightarrow{\mathsf{q}} & (E[t[r/x]], r+1) & \\
(E[\mathsf{new}\, x\, \mathsf{from}\, \mathsf{q}\, \mathsf{in}\, t], n) & \xrightarrow{\tau} & (E[t[n/x]], n+1) & (n \neq r)
\end{array}
$$

Fig. 7. Transitions between configurations

Our operational semantics will track, throughout execution, access to resources. It is defined via two sets of rules: standard (small-step) call-by-value reduction rules and a collection of rules for tracking resource usage. First we present the rules for plain call-by-value computation. For the rest of this section, we fix a function definition $\mathcal{D}$, as it is not affected by reduction.

6.1.1 Call-By-Value Reduction. **Values** are terms of one of the forms: $\mathtt{tt}$, $\mathtt{ff}$, skip, $\lambda\langle x_1, \ldots, x_n\rangle.\, t$; or *resources*, represented by natural numbers. We use $v, w, \ldots$ to range over values, or r in case the value is a resource. The call-by-value **evaluation contexts** are generated as described below.

$$
\begin{array}{rcl}
E[] & ::= & [] \mid (E[])\ \langle t_i \mid i \in I\rangle \mid E[]; t \mid (\lambda\langle x_1, \ldots, x_n\rangle.\, t)\ \langle v_1, \ldots, v_i, E[], t_{i+2}, \ldots, t_n\rangle \mid \\
& & f_i\ \langle v_1, \ldots, v_j, E[], t_{j+2}, \ldots, t_{p_i}\rangle \mid \mathsf{if}\ E[]\ u_1\ u_2 \mid \mathsf{use}_a\ E[]
\end{array}
$$

Recursively defined variables f_i are treated as values – by definition of our programs, we know that they are always bound to non-empty abstractions, *i.e.* values. Treating them as values forces the evaluation of arguments before substitution of $\mathcal{D}(f_i)$ for f_i. This will later allow for a closer match with the CPS translation.

In Figure 6 we present the rules for pure call-by-value reduction, not taking into account resource usage and non-deterministic choice. Keep in mind that these rules operate on *programs*, *i.e.* terms $\mathcal{E}; _ \vdash P : B$ with B a base type. We write $\rightsquigarrow_v^{\mathcal{D}}$ for the union of $\rightsquigarrow_v$ and $\rightsquigarrow^{\mathcal{D}}$.

6.1.2 Resource Usage. Now, we enrich our operational semantics to track resource usage. As in [Tsukada and Kobayashi 2014], resources can be dynamically created and the properties we aim to verify are resource-conscious: each resource, taken separately, must be accessed according to $\mathcal{A}$. To track usage of the rth initialized resource we build a labelled transition system $\mathfrak{L}(r)$.

Definition 43. Fix $\mathcal{D}$ and $r \in \mathbb{N}$. The **configurations** of $\mathfrak{L}(r)$ are pairs (P, n), where $n \in \mathbb{N}$ is the index of the next resource to be created and $(P, \mathcal{D})$ is a program. Transitions are given in Figure 7.

The operational semantics is well-behaved, as can be proved by an induction on P:

Lemma 44. *For each (P, n), either P is a value; or exactly one of the rules of Figures 6 and 7 applies.*

The labels of $\mathfrak{L}(r)$ are taken from $\mathsf{L} = \mathit{Act} \uplus \mathcal{Q} \uplus \{\tau\}$. For $w \in \mathsf{L}^\infty$ (possibly an infinite word on alphabet L), we write $\mathrm{tr}(w)$ for w with occurrences of τ removed. We can finally formulate:

Definition 45 (Correct program). Let $(P, \mathcal{D})$ be an $\mathcal{L}$-program and $r \in \mathbb{N}$ be a tracked resource. A maximal (possibly infinite) reduction sequence ρ in $\mathfrak{L}(r)$

$$\rho = (P, 0) \xrightarrow{v_1} (P_1, N_1) \xrightarrow{v_2} \ldots \xrightarrow{v_i} (P_i, N_i) \xrightarrow{v_{i+1}} \ldots$$

is **correct** either if $\mathrm{tr}(\rho) = \mathrm{tr}(v_1 v_2 \dots v_i \dots) = \varepsilon$ (the resource tracked is never initialized), or $\mathrm{tr}(\rho) = \mathrm{q} a_1 a_2 \dots a_j \dots$, where $a_1 a_2 \dots a_j \dots \in \mathsf{L}_{\mathrm{q}}(\mathcal{A})$. The program $(P, \mathcal{D})$ is called **correct** if, for all $r \in \mathbb{N}$, any reduction sequence in $\mathfrak{L}(r)$ starting from $(P, 0)$ is correct.

Definition 46 (Resource correctness problem). The **resource correctness problem** asks whether a given $\mathcal{L}$-program is correct.

Next we state the main result of this section, about the complexity of the resource correctness problem in our setting. The next subsection will be devoted to its proof, by a linear CPS translation into the λY-calculus.

The **depth** of a type of our language is defined by $\mathrm{depth}(\mathsf{B}) = \mathrm{depth}(\mathsf{U}) = \mathrm{depth}(\mathcal{R}) = 0$, and

$$\mathrm{depth}(A_1 \times \cdots \times A_n \Rightarrow B) = \max(\mathrm{depth}(A_1), \dots, \mathrm{depth}(A_n), \mathrm{depth}(B)) + 1.$$

The **depth** of a program $(P, \mathcal{D})$ is the maximal depth of the type of a subterm of P or of a term occurring in $\mathcal{D}$. In the remainder of the section, we will prove:

Theorem 47. *The resource correctness problem is n-EXPTIME-complete for programs of depth n.*

6.2 Linear CPS Translation

We now detail the CPS translation from $\mathcal{L}$ to the λY-calculus, which accommodates the translation better than LHORS. Thanks to the translations between LHORS and λY, we will be able to take advantage of Theorem 34, though.

Our translation exploits the observation that in the CPS of call-by-value into call-by-name, the continuations carried around are actually linearly used, in the sense of [Berdine et al. 2002]. Hence we can exploit the linearity offered by our target language, and give the terms obtained by translation a more precise kinding. Technically, our translation is a variant of the *linear CPS* (e.g. [Laird 2005]) extended with products and recursive definitions.

6.2.1 Translation of Terms. The *types* of $\mathcal{L}$ are translated to types of the λY-calculus as follows.

$$\mathsf{B}^* = o \,\&\, o \multimap o \qquad \text{and} \qquad \mathsf{U}^* = o \multimap o \qquad \text{and} \qquad \mathcal{R}^* = o \multimap o$$
$$((A_1 \times \cdots \times A_n) \Rightarrow B)^* = A_1^* \to \cdots \to A_n^* \to (B^* \to o) \multimap o$$

To all terms $\mathcal{E}; \Gamma \vdash_{\mathcal{L}} t : A$, the translation will associate a Y-free term of the λY-calculus

$$\mathcal{E}^*, \Gamma^* \mid _ \vdash t^* :: (A^* \to o) \multimap o.$$

Note that the translation of a term will not have any free linear variables, though it does have some linear kinding. In Figure 8, we give the translation for the basic λ-calculus primitives. For the resource primitives, we fix the tree signature to be Σ, defined as:

$$\{\oplus :: o \,\&\, o \multimap o,\ \mathsf{e} :: o,\ \mathsf{t} :: o \multimap o,\ \mathsf{nt} :: o \multimap o\} \cup \{\nu^{\mathsf{q}} :: o \,\&\, o \multimap o \mid \mathsf{q} \in Q\} \cup \{a :: o \multimap o \mid a \in Act\}$$

To any program $(P, \mathcal{D})$ we will associate a λY-term of ground type in context Σ, whose Böhm tree is a representation of the tree of all executions of P. The binary constructor $\oplus$ will track nondeterministic choice and each action $a \in Act$ will be recorded via a unary constructor $a :: o \multimap o$. The terminal e will be used to terminate branches, corresponding to completed computations. Finally, the remaining primitives ν^q, t, nt are used to handle dynamic creation of resources, following a trick from [Kobayashi 2009]. Each creation of a resource will be recorded as an occurrence of ν^q in the tree. The constructor ν^q is binary. In a tree generated by (the CPS translation of) a program, both sub-trees of ν^q will be *almost* identical. Their only difference will be that in the left branch, the accesses to the generated resource will be *tracked* (all accesses to that resource will be prefixed with t), whereas in the right branch, the accesses to the generated resource will be *non-tracked* (all accesses to that resource will be prefixed with nt). Later, when verifying properties of this tree, we

$$
\begin{array}{rcll}
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{tt} : \mathsf{B})^* & = & \ell k^{(o \& o \multimap o) \to o}.\, k\, (\ell x^{o \& o}.\, \pi_1\, x) & :: ((o \,\&\, o \multimap o) \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{ff} : \mathsf{B})^* & = & \ell k^{(o \& o \multimap o) \to o}.\, k\, (\ell x^{o \& o}.\, \pi_2\, x) & :: ((o \,\&\, o \multimap o) \to o) \multimap o \\
(\mathcal{E};\Gamma, x : A \vdash_{\mathcal{L}} x : A)^* & = & \ell k^{A^* \to o}.\, k\, x & :: (A^* \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{if}\, t\; u_1\; u_2 : A)^* & = & \ell k^{A^* \to o}.\, t^*\, (\lambda b^{o \& o \multimap o}.\, b\, \langle u_1^*\, k, u_2^*\, k \rangle) & :: (A^* \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{skip} : \mathsf{U})^* & = & \ell k^{(o \multimap o) \to o}.\, k\, (\ell x^{o}.\, x) & :: ((o \multimap o) \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} t; u : A)^* & = & \ell k^{A^* \to o}.\, t^*\, (\lambda x^{o \multimap o}.\, x\, (u^*\, k)) & :: (A^* \to o) \multimap o
\end{array}
$$

$$
\begin{array}{rcl}
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \lambda\langle x_1, \dots, x_n \rangle.\, t : A_1 \times \cdots \times A_n \Rightarrow B)^* & = & \ell k.\, k\, (\lambda x_1^{A_1^*} \dots \lambda x_n^{A_n^*}.\, t^*) \\
& & :: ((\overrightarrow{A_i^*} \to (B^* \to o) \multimap o) \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} t\, \langle u_i \mid 1 \le i \le n \rangle : B)^* & = & \ell k^{B^* \to o}.\, t^* \\
& & \quad (\lambda l^{A_1^* \to \cdots \to A_n^* \to (B^* \to o) \multimap o}. \\
& & \qquad u_1^*\, (\lambda x_1^{A_1^*}. \\
& & \qquad\quad \dots \\
& & \qquad\qquad u_n^*\, (\lambda x_n^{A_n^*}.\, l\, x_1\, \dots\, x_n\, k \\
& & \quad) \dots) \\
& & :: (B^* \to o) \multimap o \\
(\mathcal{E}, f_i : E_i; \Gamma \vdash_{\mathcal{L}} f_i\, \langle u_j \mid 1 \le j \le p_i \rangle : B_i)^* & = & \ell k^{B_i^* \to o}.\, u_1^*\, (\lambda x_1^{A_1^*}. \\
& & \qquad \dots \\
& & \qquad\quad u_{p_i}^*\, (\lambda x_{p_i}.\, f_i\, x_1\, \dots\, x_{p_i}\, k) \dots) \\
& & :: (B^* \to o) \multimap o
\end{array}
$$

Fig. 8. Linear CPS translation for non-resource primitives

$$
\begin{array}{rcll}
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} u_1 \oplus u_2 : A)^* & = & \ell k^{A^* \to o}.\, \oplus\, \langle u_1^*\, k, u_2^*\, k \rangle & :: \; (A^* \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{use}_a\, t : \mathsf{U})^* & = & \ell k^{(o \multimap o) \to o}.\, t^*\, (\lambda y^{o \multimap o}.\, y\, (a\, (k\, (\ell x^{o}.\, x)))) & :: \; ((o \multimap o) \to o) \multimap o \\
(\mathcal{E};\Gamma \vdash_{\mathcal{L}} \mathtt{new}\, x\, \mathtt{from}\, \mathtt{q}\, \mathtt{in}\, t)^* & = & \ell k^{A^* \to o}.\, \nu^{\mathtt{q}}\, \langle t^*[\mathtt{t}/x]\, k, t^*[\mathtt{nt}/x]\, k \rangle & :: \; (A^* \to o) \multimap o
\end{array}
$$

Fig. 9. Linear CPS translation for the resource primitives

will make sure to explore branches visiting the left sub-tree of a ν^q at most once, ensuring that at most one resource is tracked.

We can finally complete the CPS-translation. In Figure 9, we give the translation for all the primitives pertaining to resource access. This wraps up the definition of the translation of *terms* of our language: for each $\mathcal{E};\Gamma \vdash_{\mathcal{L}} t : A$, we obtain $\Sigma, \mathcal{E}^*, \Gamma^* \mid _ \vdash t^* :: (A^* \to o) \multimap o$.

6.2.2 Translation of Definitions and Programs. Assume we have a *definition*

$$\mathcal{D} = \{f_i = \lambda\langle x_{i,1}, \dots, x_{i,p_i} \rangle.\, t_i\}_{i \le N}$$

with types as indicated before. Note that we insisted (as in [Tsukada and Kobayashi 2014]) that in such definitions, each abstraction be non-empty, *i.e.* for all $1 \le i \le n$, $p_i \ge 1$. This means, in particular, that each term in the definition is a value. As is usual with CPS translations, translations of values have a particular shape, which can be confirmed directly from the definitions:

Lemma 48. *Let $\mathcal{E};\Gamma \vdash_{\mathcal{L}} v : A$ be a value. Then, v^* has the shape:*

$$\Sigma, \mathcal{E}^*, \Gamma^* \mid _ \vdash \ell k^{A^* \to o}.\, k\, v^\bullet :: (A^* \to o) \multimap o$$

for some $\mathcal{E}^*, \Gamma^* \mid _ \vdash v^\bullet :: A^*$.

In particular, for all $1 \le i \le N$, we have $\Sigma, \mathcal{E}^* \mid _ \vdash (\mathcal{D}(f_i))^\bullet = \lambda x_1^{A_1^*} \dots x_{p_i}^{A_{p_i}^*}.\, t_i^* :: E_i^*$. Exploiting the presence of products in $\lambda\!\!\lambda Y$, we can finally form a closed $\lambda\!\!\lambda Y$-term $\vdash \mathcal{D}^* :: \&_{1 \le i \le N} E_i^*$, defined as

$$\mathbb{Y}_{D : \&_{1 \le i \le N}\, E_i^*}.\, \langle (\mathcal{D}(f_i))^\bullet[\pi_j\, D/f_j] \mid 1 \le i \le N \rangle.$$

Finally, $(P, \mathcal{D})$ will be translated to $\Sigma \vdash (P, \mathcal{D})^* = P^*[\pi_i\, \mathcal{D}^*/f_i]\, (\lambda k^{o \multimap o}.\, k\, \mathrm{e}) :: o$.

6.2.3 Complexity of the Translation. The remainder of this section will be concerned with correctness of the translation, *i.e.* showing that the Böhm tree of the translation describes the resource usage of P adequately. Also, in order to exploit our model-checking algorithm, we need to ensure that the translation of a program of depth n has *linear order* n, bounded linear depth, and that its size is polynomial in the size of the original term. The last point is straightforward, so let us focus on the previous ones. It is easy to show that

Lemma 49. *For any type A of $\mathcal{L}$, we have* $\mathrm{depth}(A) = \ell o(A^*)$.

However, for terms there is a mismatch: a subterm $\mathcal{E}; \Gamma \vdash_{\mathcal{L}} t : A$ is interpreted as $\mathcal{E}^*, \Gamma^* \mid _ \vdash t^* :: (A^* \to o) \multimap o$. So, if a program has depth d, *i.e.* a subterm of type A with $\mathrm{depth}(A) = d$, its translation will have a subterm $t^* :: (A^* \to o) \multimap o$ of *linear order* $d + 1$. Fortunately, one can always track a subterm of maximal depth in $(P, \mathcal{D})$ to an abstraction, which can be simplified exploiting the intuition that it will never itself be used as an argument of a function.

Let d be the depth of $(P, \mathcal{D})$. To perform the simplification, we introduce an *optimized* translation operation $(-)^\circ$. On any subterm of depth strictly less than d, $(-)^\circ$ will work just as $(-)^*$. However, for $A = A_1 \times \cdots \times A_n \to B$ with $\mathrm{depth}(A) = d$, and $\mathcal{E}; \Gamma \vdash_{\mathcal{L}} t : A$, writing $\Delta = y_1 :: (A_1^* \to o) \multimap o, \dots, y_n :: (A_n^* \to o) \multimap o$, we shall have $\mathcal{E}^*, \Gamma^*, \Delta \mid k :: B^* \to o \vdash t^\circ :: o$ defined by:

$$
\begin{array}{rclrcl}
(\lambda \langle x_1, \dots, x_n \rangle.\, t)^\circ & = & y_1\, (\lambda x_1.\, \cdots\, y_n\, (\lambda x_n.\, t^*\, k) \dots) & (u_1 \oplus u_2)^\circ & = & \oplus \langle u_1^\circ, u_2^\circ \rangle \\
(\mathtt{if}\; t\; u_1\; u_2)^\circ & = & t^*\, (\lambda b.\, b\, \langle u_1^\circ, u_2^\circ \rangle) & (t\, \langle u_1, \dots, u_n \rangle)^\circ & = & \ell k^{B^* \to o}.\, t^\circ[u_i^*/y_i] \\
(t; u)^\circ & = & t^*\, (\lambda x.\, x\, u^\circ) & (\mathtt{new}\, x\, \mathtt{from}\, \mathtt{q}\, \mathtt{in}\, t)^\circ & = & \nu^{\mathtt{q}}\, \langle t^\circ[\mathtt{t}/x], t^\circ[\mathtt{nt}/x] \rangle
\end{array}
$$

where we sometimes still use $(-)^*$ to emphasize terms where the translation is not changed (because the translated term has a type of depth strictly smaller than d). In the clause for $t\, \langle u_1, \dots, u_n \rangle$, we assume that t has type A – note that this clause only covers application of a *term* to arguments; the translation of the application of a recursively defined function is not changed. This definition does not cover, *e.g.*, free variables of type A of maximal depth; but those cannot appear in a program (as they would need to be abstracted at some point, yielding a term of depth exceeding d). Thus, the following lemma can be proved by induction on t:

Lemma 50. *For any subterm $\mathcal{E}; \Gamma \vdash_{\mathcal{L}} t : A$ of $(P, \mathcal{D})$ with* $\mathrm{depth}(A) < d$, $\mathcal{E}^*, \Gamma^* \mid _ \vdash t^\circ :: (A^* \to o) \multimap o$ *has linear order less than d, size linear in that of t, and linear depth* 2. *t° and t^* are $\beta\eta$-equivalent.*

For a program $(P, \mathcal{D})$ of depth d, we change the translation of recursively defined functions to $(\mathcal{D}(f_i))^\bullet = \lambda x_1^{A_1^*} \dots x_{p_i}^{A_{p_i}^*}.\, t_i^\circ$, leaving the rest of the definition unchanged. Thus we get $(P, \mathcal{D})^\circ$, a λY-term of kind o in context Σ, $\beta\eta$-equivalent to $(P, \mathcal{D})^*$ but of order d, size linear in that of $(P, \mathcal{D})$, and linear depth 2. In the remainder of this section, we prove the correctness of the translation; hence we will work with the un-optimized translation $(-)^*$. The reader should keep in mind that $(P, \mathcal{D})^* \cong_{\beta\eta} (P, \mathcal{D})^\circ$, so any correctness result proved for one of them will hold for the other.

6.2.4 Simulation. We now set to establish a correspondence between reductions on both sides of the CPS translation. We first focus on the reduction $\rightsquigarrow_v$. This part of the proof follows standard lines, and we only point out the major steps.

First, we prove by induction on t that the translation preserves the substitution with a value.

Lemma 51. *Let $\mathcal{E}; \Gamma, x : A \vdash_{\mathcal{L}} t : B$ be any term of $\mathcal{L}$, and $\mathcal{E}; \Gamma \vdash_{\mathcal{L}} v : A$ be a value of type A. Then, $(t[v/x])^* = t^*[v^\bullet/x]$, where $v^* = \ell k^{A^* \to o}.\, k\, v^\bullet$ as given by Lemma 48.*

From that, we can easily deduce that the translation is a simulation with respect to β-reduction.

$$
\begin{array}{rclcrcl}
(\{\oplus\,\langle t_1, t_2\rangle\}, n) & \xrightarrow{\tau} & (\{t_i\}, n) & \quad & (\{\mathsf{t}\,(a\,t)\}, n) & \xrightarrow{a} & (\{t\}, n) \\
(\{\mathsf{nt}\,(a\,t)\}, n) & \xrightarrow{\tau} & (\{t\}, n) & & (\{\nu^{\mathsf{q}}\,\langle t_1, t_2\rangle\}, r) & \xrightarrow{\mathsf{q}} & (\{t_1\}, r+1)
\end{array}
$$

$$
(\{\nu^{\mathsf{q}}\,\langle t_1, t_2\rangle\}, n) \xrightarrow{\tau} (\{t_2\}, n+1) \qquad (n \neq r)
$$

Fig. 10. Transitions of $\mathfrak{L}'(r)$.

Lemma 52. *For any $\mathcal{E}; _ \vdash_{\mathcal{L}} t : A$, if there exists u such that $t \rightsquigarrow_v u$ then $t^* \triangleright_\beta^* \triangleright_\eta u^*$.*

Proof. For all $E[-]$, there is a $\lambda\!\!\lambda Y$-context $C[-]$ such that for all appropriately typed $\Gamma; _ \vdash_{\mathcal{L}} t : A$; $(E[t])^* = C[t^*]$. We check all $\rightsquigarrow_v$-redexes of Figure 6, using Lemma 51 for application. □

Now that we have established our simulation result without recursion, we add recursion to the mix. It will be useful to note that the redexes that come from the translation of CBV redexes always appear in *head position*: we define the **head contexts** in the $\lambda\!\!\lambda Y$-calculus, as follows:

$$H[-] ::= [-] \mid \lambda x.\, H[-] \mid \ell x.\, H[-] \mid H[-]\, u \mid \pi_i\, H[-]$$

Lemma 53. *Let $E[-]$ be a CBV evaluation context. Then, there exists a head context $H[-]$, such that for all $\mathcal{E}; \Gamma \vdash_{\mathcal{L}} t : A$ of the appropriate type, $(E[t])^* \triangleright_\beta^* H[t^*]$.*

Consider a program $(P, \mathcal{D})$, interpreted as in Section 6.2.2. We observe that the use of recursive calls in P matches fixpoint unfoldings in the translated term. Leveraging Lemma 53, we have:

Lemma 54. *Fix a definition $\mathcal{D}$. Then, if $P_1 \rightsquigarrow^{\mathcal{D}} P_2$, we have $(P_1, \mathcal{D})^* \cong_{\beta\eta} \triangleright_\delta \cong_{\beta\eta} (P_2, \mathcal{D})^*$, where $\cong_{\beta\eta}$ is $\beta\eta$-equivalence.*

From there, we can finally prove the adequacy of the translation.

Proposition 55 (Adequacy). *For $(P, \mathcal{D})$ a program, $\rightsquigarrow_v^{\mathcal{D}}$ terminates on P iff $(P, \mathcal{D})^*$ is solvable.*

Proof. The translation is designed so that recursion matches on both sides of the translations: precisely, the translation induces a bisimulation between the relation $\rightsquigarrow_v^* \rightsquigarrow^{\mathcal{D}}$ on programs-in-definition, and relation $\triangleright_\delta$ on $\beta\eta$-equivalence classes of $\lambda\!\!\lambda Y$-terms.

That it is a simulation is Lemma 54. Assume $(P, \mathcal{D})^* \cong_{\beta\eta} \triangleright_\delta$, thus the $\beta\eta$-normal form of $(P, \mathcal{D})^*$ has a $\mathbb{Y}_x.\, t$ subterm in head position. Reduce P via $\rightsquigarrow_v^*$ as much a possible – this terminates. If it terminates on anything else than a $\rightsquigarrow^{\mathcal{D}}$-redex, then the $\beta\eta$-normal form of the translation has a variable in head position, contradicting that it is a $\triangleright_\delta$-redex. So it is indeed a $\rightsquigarrow^{\mathcal{D}}$-redex, and necessarily firing it translates to firing the unique $\triangleright_\delta$-redex of (the normal form of) $(P, \mathcal{D})^*$.

The proposition follows easily from this bisimulation. □

6.2.5 Resource Verification as $\lambda\!\!\lambda Y$ Model-Checking. Finally, using the adequacy of the translation, we study its action on the resource usage primitives. For that, we construct a second LTS, this time from $\lambda\!\!\lambda Y$-terms $\Sigma \mid _ \vdash t :: o$, matching $\mathfrak{L}(r)$ from Definition 43. Its **configurations** are pairs $(\mathtt{t}, n)$ where $\mathtt{t}$ is a $\beta\eta\delta$-equivalence class of $\lambda\!\!\lambda Y$-terms (we will often write such equivalence classes as $\{t\}$, for t a representative), and where as before, $n \in \mathbb{N}$ represents the next resource index to be initialized. The **transitions**, labeled again by L, are given in Figure 10. We write $\mathfrak{L}'(r)$ for this LTS.

Paths in this LTS starting from $(\{t\}, 0)$ perform an exploration of $\mathrm{BT}(t)$ (with the proviso that $\mathrm{BT}(t)$ is such that each occurrence of a is preceeded by t or nt, which will be true in the translation), branching non-deterministically at occurrences of $\oplus$, and turning left *exactly once* at occurrences of ν^{q}: when the required index has been reached. $\mathfrak{L}'(r)$ satisfies the following key property.

Proposition 56. *For any $r \in \mathbb{N}$, the relation $\mathcal{B}$ between configurations of $\mathfrak{L}(r)$ and $\mathfrak{L}'(r)$ having*

$$
\begin{array}{lcll}
(P, n) & \mathcal{B} & (\{(P, \mathcal{D})^*[\mathsf{nt}/i]\}, n) & (n \leq r) \\
(P, n) & \mathcal{B} & (\{(P, \mathcal{D})^*[\mathsf{t}/r][\mathsf{nt}/i \mid i \neq r]\}, n) & (\textit{otherwise})
\end{array}
$$

for any $\mathcal{E}; 0 : \mathcal{R}, \ldots, (n-1) : \mathcal{R} \vdash_{\mathcal{L}} P : B$ *with* B *a base type, is a weak bisimulation.*

Proof. From the definition of translation and using Lemma 53, it is a simulation. The other direction follows by Proposition 55: if (P, n) $\mathcal{B}$ $(\{t\}, n)$ and $\{t\}$ makes one of the transitions of Figure 10, then t is solvable. Hence, $\rightsquigarrow_v^{\mathcal{D}}$ terminates on P; eventually reducing to Q. But from Lemma 44, either Q is a value (absurd or that would contradict that $\{t\}$ makes a transition), or a rule from Figure 7 applies – but only the rule matching the transition of $\{t\}$ is possible. □

For every closed $\mathcal{E}; _ \vdash_{\mathcal{L}} P : B$ with B base type, we have $(P, 0)$ $\mathcal{B}$ $((P, \mathcal{D})^*, 0)$, so the trees of $\mathfrak{L}(r)$ and $\mathfrak{L}'(r)$ respectively reachable from those are weakly bisimilar: their maximal branches have the same traces. By definition of $\mathfrak{L}(r)$, P is correct if for all $r \in \mathbb{N}$, the maximal branches of $\mathfrak{L}(r)$ have correct trace. Likewise, by definition of $\mathfrak{L}'(r)$, its maximal branches correspond to the maximal branches of $\mathrm{BT}((P, \mathcal{D})^*)$ taking the left sub-tree of a v^{q} exactly once, when encountering a v^{q} for the r-th time. Hence, P is correct iff all the branches of $\mathrm{BT}((P, \mathcal{D})^*)$ visiting the left subtree of a v^{q} *exactly once* are correct. Following [Kobayashi and Ong 2011], from $\mathcal{A}$ we may construct a disjunctive automaton $\mathcal{A}'$ that accepts trees that have a branch visiting the left subtree of v^{q} exactly once, and where the sequence of tracked resources used *fail* $\mathcal{A}$.

Corollary 57. *For any program* $(P, \mathcal{D})$ *and deterministic parity automaton* $\mathcal{A}$ *there exist*

- *a* λY*-term* $\Sigma \mid _ \vdash (P, \mathcal{D})^\circ :: o$ *of linear order equal to the depth of* $(P, \mathcal{D})$*, of size linear in that of* $(P, \mathcal{D})$*, and linear depth* 2*,*
- *a linear-non-linear APTA* $\mathcal{A}'$ *of size linear in that of* $\mathcal{A}$

such that P *is correct iff* $\mathcal{A}'$ *rejects* $\mathrm{BT}((P, \mathcal{D})^\circ)$*.*

The construction of $\mathcal{A}'$ from $\mathcal{A}$ proceeds as in [Kobayashi and Ong 2011]. Theorem 47 now follows.

7 FURTHER DIRECTIONS

Our results demonstrate the potential of λY and LHORS to serve a unifying framework for studies into higher-order verification. The significant body of literature on connections between linear logic and programming languages opens up many avenues for future work. Verification algorithms for LHORS have, by design, quite a wide application range, and the translations we considered exploit linearity to a small extent only.

Some obvious and short-term directions beyond model-checking include global model-checking [Broadbent et al. 2010] and selection [Carayol and Serre 2012], which we expect (as in the standard case) to enjoy the same complexity, with respect to the same measures.

Amongst other more open-ended questions, we would like to understand better the additional expressiveness contributed by the addition of linear kinds. What new infinite trees can we get, while retaining efficient verification algorithms? What is the trade-off between the complexity boost coming from linear typing and properties expressible by LNAPTAs?

Finally, it would be interesting to explore the practical impact of our work, for instance, by using type inference to obtain more refined linear-non-linear typings for functional programs, so as to improve the complexity of verification.

ACKNOWLEDGMENTS

The second author acknowledges the support of LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR), and the ANR Project RAPIDO, ANR-14-CE25-0007. The research also benefited from the support of the London Mathematical Society.

REFERENCES

K. Aehlig. 2007. A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of Automata. *Logical Methods in Computer Science* 3, 3 (2007).

A. Barber and G. Plotkin. 1996. *Dual Intuitionistic Linear Logic.* Technical Report LFCS-96-347. LFCS, Division of Informatics, University of Edinburgh.

J. Berdine, P. W. O'Hearn, U. S. Reddy, and H. Thielecke. 2002. Linear Continuation-Passing. *Higher-Order and Symbolic Computation* 15, 2-3 (2002), 181–208.

C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. 2013. C-SHORe: a collapsible approach to higher-order verification. In *Proceedings of ICFP.* 13–24.

C. H. Broadbent, A. Carayol, C.-H. L. Ong, and O. Serre. 2010. Recursion Schemes and Logical Reflection. In *Proceedings of LICS.* 120–129.

A. Carayol and O. Serre. 2012. Collapsible Pushdown Automata and Labeled Recursion Schemes: Equivalence, Safety and Effective Selection. In *Proceedings of LICS.* 165–174.

W. Damm. 1977. Higher type program schemes and their tree languages. In *Theoretical Computer Science, 3rd GI-Conference, Darmstadt, Germany, March 28-30, 1977, Proceedings (Lecture Notes in Computer Science)*, Vol. 48. Springer, 51–72.

J.-Y. Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

C. Grellois. 2016. *Semantics of linear logic and higher-order model-checking.* Ph.D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01311150

C. Grellois and P.-A. Melliès. 2015a. Finitary Semantics of Linear Logic and Higher-Order Model-Checking. In *Proceedings of MFCS (Lecture Notes in Computer Science)*, Vol. 9234. Springer, 256–268.

C. Grellois and P.-A. Melliès. 2015b. Relational Semantics of Linear Logic and Higher-order Model Checking. In *Proceedings of CSL.* 260–276.

A. Haddad. 2013. *Shape-Preserving Transformations of Higher-Order Recursion Schemes.* Thèse de Doctorat. Université Paris Diderot - Paris 7.

M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. 2008. Collapsible Pushdown Automata and Recursion Schemes. In *Proceedings of LICS.* Computer Society Press, 452–461.

M. Hofmann and J. Ledent. 2017. A cartesian-closed category for higher-order model checking. In *Proceedings of LICS.* 1–12.

M. Jurdziński. 2000. Small Progress Measures for Solving Parity Games. In *Proceedings of STACS.* 290–301.

N. Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of POPL.* 416–428.

N. Kobayashi and C.-H. L. Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *Proceedings of LICS.*

N. Kobayashi and C.-H. L. Ong. 2011. Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus. *Logical Methods in Computer Science* 7, 4 (2011).

N. Kobayashi, R. Sato, and H. Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of PLDI.* 222–233.

N. Kobayashi, N. Tabuchi, and H. Unno. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of POPL.* 495–508.

J. Laird. 2005. Game semantics and linear CPS interpretation. *Theor. Comput. Sci.* 333, 1-2 (2005), 199–224.

P.-A. Melliès. 2014. Linear logic and higher-order model-checking. Talk at Institut Henri Poincaré, http://www.pps.univ-paris-diderot.fr/~mellies/slides/workshop-IHP-model-checking.pdf.

A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno. 2016. Temporal verification of higher-order functional programs. In *Proceedings of POPL.* ACM, 57–68.

R. P. Neatherway, S. J. Ramsay, and C.-H. L. Ong. 2012. A traversal-based algorithm for higher-order model checking. In *Proceedings of ICFP.* 353–364.

M. Nivat. 1972. On the interpretation of recursive program schemes. In *Symposia Matematica.*

C.-H. L. Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *Proceedings of LICS.* Computer Society Press, 81–90.

G. Pottinger. 1981. The Church-Rosser theorem for the typed λ-calculus with surjective pairing. *Notre Dame J. Formal Logic* 22 (3) (1981), 264–268.

S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. 2014. A type-directed abstraction refinement approach to higher-order model checking. In *Proceedings of POPL.* 61–72.

S. Salvati and I. Walukiewicz. 2015. A Model for Behavioural Properties of Higher-order Programs. In *Proceedings of CSL (LIPIcs)*, Vol. 41. 229–243.

S. Salvati and I. Walukiewicz. 2016. Simply typed fixpoint calculus and collapsible pushdown automata. *Mathematical Structures in Computer Science* 26, 7 (2016), 1304–1350.

O. Serre. 2013. *Playing with Trees and Logic*. Habilitation à Diriger des Recherches. Université Paris Diderot - Paris 7. http://www.liafa.univ-paris-diderot.fr/~serre/papers/HDR.pdf

R. Statman. 2004. On the lambda$Y$ calculus. *Ann. Pure Appl. Logic* 130, 1-3 (2004), 325–337.

T. Tsukada and N. Kobayashi. 2010. Untyped Recursion Schemes and Infinite Intersection Types. In *Proceedings of FOSSACS (Lecture Notes in Computer Science)*, Vol. 6014. Springer, 343–357.

T. Tsukada and N. Kobayashi. 2014. Complexity of Model-Checking Call-by-Value Programs. In *Proceedings of FOSSACS'14*. 180–194.