Slightly Non-Linear Higher-Order Tree Transducers

Lê Thành Dũng (Tito) Nguyễn
CNRS & Aix-Marseille University, France

Gabriele Vanoni
IRIF, Université Paris Cité, France

Abstract

We investigate the tree-to-tree functions computed by "affine λ-transducers": tree automata whose memory consists of an affine λ-term instead of a finite state. They can be seen as variations on Gallot, Lemay and Salvati's Linear High-Order Deterministic Tree Transducers.

When the memory is almost purely affine (*à la* Kanazawa), we show that these machines can be translated to tree-walking transducers (and with a purely affine memory, we get a reversible tree-walking transducer). This leads to a proof of an inexpressivity conjecture of Nguyễn and Pradic on "implicit automata" in an affine λ-calculus. We also prove that a more powerful variant, extended with preprocessing by an MSO relabeling and allowing a limited amount of non-linearity, is equivalent in expressive power to Engelfriet, Hoogeboom and Samwel's invisible pebble tree transducers.

The key technical tool in our proofs is the Interaction Abstract Machine (IAM), an operational avatar of Girard's geometry of interaction, a semantics of linear logic. We work with ad-hoc specializations to λ-terms of low exponential depth of a tree-generating version of the IAM.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Transducers

Keywords and phrases Almost affine lambda-calculus, geometry of interaction, reversibility, tree transducers, tree-walking automata

Digital Object Identifier 10.4230/LIPIcs.STACS.2025.68

Related Version *Technical report with appendix*: `https://arxiv.org/abs/2402.05854` [38]

Funding *Lê Thành Dũng (Tito) Nguyễn*: Supported by the DyVerSe project (ANR-19-CE48-0010). *Gabriele Vanoni*: Supported by the ANR PPS Project (ANR-19-CE48-0014) and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101034255.

Acknowledgements We thank Damiano Mazza and Cécilia Pradic for discussions on the possible applications of the Geometry of Interaction to implicit complexity and automata theory.

1 Introduction

This paper investigates the expressive power of various kinds of *tree transducers*: automata computing tree-to-tree functions. This is a topic with a long history, and many equivalences between machine models are already known. For instance, the class of monadic second-order transductions[1] (MSOTs), whose name refers to a definition by logic, is also captured by various tree transducer models (e.g. [19]) or by a system of primitive functions and combinators [10]. This class is closed under composition, and includes functions such as:

mirror and add a d above each b:	$a(b(c), c) \mapsto a(c, d(b(c)))$
relabel each c by parity of its depth:	$a(b(c), c) \mapsto a(b(0), 1)$
count number of non-a nodes in unary:	$a(b(c), c) \mapsto S(S(S(0)))$

[1] We consider only tree-to-tree transductions here, but there is a rich theory of MSOTs between graphs, or between arbitrary relational structures, cf. [12]. See also [8] for a history of MSO transductions. In the well-studied special case of string functions, MSOTs are called "regular functions", cf. [33].

42nd International Symposium on Theoretical Aspects of Computer Science (STACS 2025).
Editors: Olaf Beyersdorff, Michał Pilipczuk, Elaine Pimentel, and Nguyễn Kim Thắng;
Article No. 68; pp. 68:1–68:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Some machines for MSOTs, such as the restricted macro tree transducers[2] of [20], involve *tree contexts* as data structures used in their computation. A tree context is a tree with "holes" at some leaves – for example $b(a(c, a(\langle\cdot\rangle, c)))$ – and these holes are meant to be substituted by other trees. Thus, it represents a simple function taking trees to trees: these transducers use functions as data. From this point of view, in the spirit of functional programming, it also makes sense to consider transducers manipulating *higher-order data*, that is, functions that may take functions as arguments (a function of order $k+1$ takes arguments of order at most k). This idea goes back to the 1980s [21] (cf. Remark 2.9), and a recent variant by Gallot, Lemay and Salvati [22, 23] computes exactly the tree-to-tree MSOTs.

Linear/affine λ-calculus for MSOTs. Gallot et al. [22, 23] use the λ-*calculus* to represent higher-order data, and in order to control the expressive power, they impose a *linearity* restriction on λ-terms. In programming language theory, a *linear* function uses its argument *exactly once*, while an *affine* function must use its argument *at most once* – affineness thus mirrors the "single-use restrictions" that appear in various tree transducer models [7, 20, 4, 10].

An independent but similar characterization of MSOTs appeared around the same time in Nguyễn and Pradic's work on "implicit automata" [37, 34]. In an analogous fashion to how the untyped λ-calculus can be used as a Turing-complete programming language, they consider typed λ-terms seen as standalone programs. For a well-chosen type system – which enforces a linearity restriction – and input/output convention, it turns out that the functions computed by these λ-terms are exactly the MSO tree transductions [34, Theorem 1.2.3].

In fact, the proof of [34, Theorem 1.2.3] introduces, as an intermediate step, a machine model – the "single-state $\mathfrak{L}$-BRTTs" of [34, Section 6.4] – that uses λ-terms as memory to capture the class of MSOTs. Clearly, this device is very close to the tree transducer model of Gallot et al. Yet there are also important differences between the two: they can be understood as two distinct ways of extending what we call "purely linear λ-transducers", as we will explain. Actually, to avoid some uninteresting pathologies (cf. [34, Theorem 7.0.2]), we shall prefer to work with affine λ-transducers instead.

▶ **Example 1.1.** Purely affine λ-transducers will be properly defined later, but for now, for the sake of concreteness, let us exhibit one such transducer. It takes input trees with binary a-labeled nodes, unary b-labeled nodes and c-labeled leaves, and is specified by the λ-terms:

$$t_a = \lambda\ell.\ \lambda r.\ \lambda x.\ \ell\ (r\ x) \qquad t_b = \lambda f.\ \lambda x.\ S\ (f\ x) \qquad t_c = S \qquad u = \lambda f.\ f\ 0$$

where S and 0 are constants from the output alphabet. The input $a(b(c), c)$ is then mapped to $u\ (t_a\ (t_b\ t_c)\ t_c)$ which evaluates to the normal form $S\ (S\ (S\ 0))$. So this λ-transducer computes the aforementioned "number of non-a nodes written in unary" function. (It actually is purely linear: each bound variable has exactly one occurrence.)

The issue with this transducer model is its lack of expressiveness, as shown by the following consequence of our results, which settles an equivalent[3] conjecture on "implicit automata" that had been put forth by Nguyễn and Pradic in [37, Section 5.3]. (We will come back in Remark 7.1 to how they overcome this to characterize tree-to-tree MSO transductions.)

[2] Which are very similar to some other transducer models for MSOTs: the bottom-up ranked tree transducers of [4, Sections 3.7–3.8] and the register tree transducers of [10, Section 4].

[3] A routine syntactic analysis akin to [37, Lemma 3.6] shows that it is indeed equivalent. Note that the results of [37] are indeed about affine, not linear, λ-calculi.

▶ **Corollary 1.2** (of Theorem 1.4 below). *There exists a regular tree*[4] *language whose indicator function cannot be computed by any purely affine λ-transducer.*

Gallot et al. avoid this limitation by extending the transducer model with common automata-theoretic features. They then show [23, Chapter 7] that the usual linear λ-terms lead to the previously mentioned characterization of MSOTs, while relaxing the linearity condition to "almost linearity" yields the larger class of "MSOTs with sharing"[5] (MSOT-S).

This notion of *almost linear λ-term* was introduced by Kanazawa [28], who also studied the almost affine case in [28, 27]. In fact, the aforementioned characterization of MSOT-S was first claimed by Kanazawa in a talk more than 15 years ago [26], although he never published a proof to the best of our knowledge.

Flavors of affine types. We shall work with an affine type system that allows some data to be marked as duplicable via the exponential modality "!". The grammar of types is thus $A, B ::= o \mid A \multimap B \mid !A$ – the connective $\multimap$ is the affine function arrow.

The point is to allow us to restrict duplication by means of syntactic constraints on '!'.

▶ **Definition 1.3.** *A* purely affine *type does not contain any '!', i.e. is built from o and $\multimap$.*

A type is said to be almost purely affine *when the only occurrences of '!' are applied to o. In particular, every purely affine type is almost purely affine.*

For example, $(o \multimap !o) \multimap !o$ is almost purely affine but not $!(o \multimap o)$.

The latter definition is motivated by the aforementioned almost affine λ-terms [27], which allow the variables of base type o to be used multiple times, e.g. $\lambda y.\ \lambda f.\ \lambda g.\ (\lambda x.\ f\ x\ x)\ (g\ y\ y)$ is almost affine (and even almost linear) for $x : o$ and $y : o$. Inconveniently, almost linear/affine λ-terms are not closed under β-reduction, as remarked in [27, §4]. For instance, the previous term reduces to $\lambda y.\ \lambda f.\ \lambda g.\ f\ (g\ y\ y)\ (g\ y\ y)$ which uses $g : o \multimap o \multimap o$ twice. The "!" modality provides a convenient way to realize a similar idea while avoiding this drawback.

Each λ-transducer has in its definition a *memory type*, which is for instance $o \multimap o$ for Example 1.1. We extend Definition 1.3 to λ-transducers: a λ-transducer is purely (resp. almost purely) affine when its memory type is purely (resp. almost purely) affine.

Contributions. First, we study (almost) purely affine λ-transducers, relating them to yet another machine model: *tree-walking tree transducers*, see e.g. [19]. Those are devices with a finite-state control and a reading head moving around the nodes of the input tree; in one step, the head can move to the parent or one of the children of the current node.

▶ **Theorem 1.4.** *Writing $\subseteq$ for a comparison in expressive power, we have:*

$$\textit{purely affine } \lambda\textit{-transducer} \subseteq \textit{reversible tree-walking transducer}$$
$$\textit{almost purely affine } \lambda\textit{-transducer} \subseteq \textit{tree-walking transducer}$$

Corollary 1.2 then follows immediately from a result of Bojańczyk and Colcombet [9]: there exists a regular tree language not recognized by any tree-walking automaton.

[4] This phenomenon depends on using trees as inputs. Over strings, purely affine λ-terms can be used to recognize any regular language [37, Theorem 5.1].

[5] An MSOT-S can be decomposed as an MSO tree-to-graph transduction (Footnote 1), that produces a rooted directed acyclic graph (DAG), followed by the unfolding of this graph into a tree. The rooted DAG is a compressed representation of the output tree with some "sharing" of subtrees.

We also obtain characterizations of MSOTs and of MSOT-Ss, by preprocessing the input with an *MSO relabeling* – a special kind of MSOT that can only change the node labels in a tree, but keeps its structure as it is. Morally, this preprocessing amounts to the same thing as the automata-theoretic features – finite states and regular look-ahead – used in [22, 23].

▶ **Theorem 1.5.** *Using $\equiv$ to denote an equivalence in expressive power, we have:*

$$\textit{purely affine } \lambda\textit{-transducer} \circ \textit{MSO relabeling} \equiv \textit{MSOT}$$
$$\textit{almost purely affine } \lambda\textit{-transducer} \circ \textit{MSO relabeling} \equiv \textit{MSOT-S}$$

This should be informally understood as a mere rephrasing of the results of Gallot et al. and of Kanazawa, except with affine rather than linear λ-terms. On a technical level, we derive our right-to-left inclusions from their results; there turns out to be a minor mismatch, and working with affineness rather than linearity proves convenient to overcome it. As for the left-to-right inclusions, we get them as corollaries of Theorem 1.4.

Finally, we give a genuinely new characterization of the class MSOT-S^2 of functions that can be written as *compositions of two MSO transductions with sharing.*

▶ **Definition 1.6.** *A type is* almost !-depth 1 *when the only occurrences of '!' are applied to almost purely affine types (e.g.* $!(!o \multimap o)$ *is almost !-depth 1, but not* $!!(o \multimap o)$*).*

▶ **Theorem 1.7.** *Almost !-depth 1 λ-transducer $\circ$ MSO relabeling $\equiv$ MSOT-S^2.*

To prove the left-to-right inclusion, we compile these λ-transducers to *invisible pebble tree transducers* [18] – an extension of tree-walking transducers known to capture MSOT-S^2. For the converse, we rely on a simple composition procedure for λ-transducers, which gives us:

▶ **Proposition 1.8.** *Suppose that the tree-to-tree functions f and g are computed by λ-transducers with respective memory types A and B. Then $g \circ f$ is computed by a λ-transducer with memory type $A\{o := B\}$.*

Key tool: the Interaction Abstract Machine (IAM). To translate λ-transducers into tree-walking or invisible pebble tree transducers, we use a mechanism that evaluates a λ-term using a pointer to its syntax tree that is moved by local steps – just like a tree-walking reading head. This mechanism, called the Interaction Abstract Machine, was derived from a family of semantics of linear λ-calculi known as "geometry of interaction" (GoI) – for more on its history, see [46, Chapter 3]. Our approach thus differs from both the proofs of Gallot et al. in [23], which go through a syntactic procedure for lowering the order of λ-terms, and those of Nguyễn and Pradic in [34], which rely on another kind of denotational semantics.

The IAM satisfies a well-known *reversibility* property, see e.g. [46, Proposition 3.3.4] – it even appears in the title of the seminal paper [14]. In the purely affine case, this gives us reversible tree-walking transducers in Theorem 1.4 – these have not appeared explicitly in the literature, but we define them similarly to the existing reversible graph-walking automata [39] and reversible two-way string transducers [15]. In the almost purely affine and almost !-depth 1 cases, we use an ad-hoc optimization of the IAM that breaks reversibility.

Preliminaries on trees. A *ranked alphabet* Σ is a finite set with a "rank" (or "arity") $\mathrm{rk}(c) \in \mathbb{N} = \{0, 1, \dots\}$ associated to each "letter" $c \in \Sigma$. It can be seen as a first-order signature of function symbols, and by a *tree* over Σ we mean a closed first-order term over this signature. For instance, a tree over $\{a : 2,\ b : 1,\ c : 0\}$ may have binary a-labeled nodes, unary b-labeled nodes and c-labeled leaves; examples include $a(b(c), c)$ or $b(a(a(c, c), c))$.

We write $\mathrm{Tree}(\Sigma)$ for the set of trees over the ranked alphabet Σ. It will also be convenient to work with the sets $\mathrm{Tree}(\Sigma, X) = \mathrm{Tree}(\Sigma \cup \{x : 0 \mid x \in X\})$ of trees over Σ with the additional possibility of having X-labeled leaves.

2 Affine λ-terms and tree-to-tree λ-transducers

The grammar of our λ-terms, where a ranges over constants and x over variables, is

$$t, u ::= a \mid x \mid \lambda x.\, t \mid t\ u \mid !t \mid \texttt{let}\ !x = u\ \texttt{in}\ t$$

These λ-terms are considered up to renaming of bound variables ($\lambda x.\, t$ binds x in t, while $\texttt{let}\ !x = u\ \texttt{in}\ t$ binds x in t but not in u). We will also use *contexts*, which are λ-terms containing one occurrence of a special symbol, the hole $\langle\cdot\rangle$:

$$C, D, E ::= \langle\cdot\rangle \mid \lambda x.\, C \mid t\ C \mid C\ u \mid !C \mid \texttt{let}\ !x = C\ \texttt{in}\ t \mid \texttt{let}\ !x = u\ \texttt{in}\ C$$

Plugging, i.e. substituting the hole of a context C for a term t, potentially capturing free variables, is written $C\langle t\rangle$. For example, if $C = \lambda x.\, t\ \langle\cdot\rangle$, then $C\langle u\ v\rangle = \lambda x.\, t\ (u\ v)$. The linear logic tradition calls a term of the form $!t$ a *box*, and the number of nested boxes surrounding a subterm of some term is called its *depth* within this term. Similarly, the *depth of a context* C is defined to be the number of boxes surrounding the hole $\langle\cdot\rangle$. In this paper, however, we are counting only boxes which do *not* surround terms of the base type o.

Typing rules. Our type system is a variant of Dual Intuitionistic Linear Logic [6], with weakening to make it affine. As already said, our grammar of types is $A, B ::= o \mid A \multimap B \mid !A$. As usual, $\multimap$ is right-associative: the parentheses in $A \multimap (B \multimap C)$ can be dropped.

The typing contexts are of the form (unrestricted variables) | (affine variables). In the rules below, a comma between two sets of typed affine variables denotes a disjoint union; this corresponds to prohibiting the use of the same affine free variable in two distinct subterms.

$$\frac{}{\Theta \mid \Phi, x : A \vdash x : A} \qquad \frac{\Theta \mid \Phi, x : A \vdash t : B}{\Theta \mid \Phi \vdash \lambda x.\, t : A \multimap B} \qquad \frac{\Theta \mid \Phi \vdash t : A \multimap B \quad \Theta \mid \Phi' \vdash u : A}{\Theta \mid \Phi, \Phi' \vdash t\ u : B}$$

$$\frac{}{\Theta, x : A \mid \Phi \vdash x : A} \qquad \frac{\Theta \mid \varnothing \vdash t : A}{\Theta \mid \varnothing \vdash !t : !A} \qquad \frac{\Theta \mid \Phi \vdash u : !A \quad \Theta, x : A \mid \Phi' \vdash t : B}{\Theta \mid \Phi, \Phi' \vdash \texttt{let}\ !x = u\ \texttt{in}\ t : B}$$

We also work with constants whose types are fixed in advance. Fixing $a : A$ means that we have the typing rule $\frac{}{\Theta \mid \Phi \vdash a : A}$. We shall also abbreviate $\varnothing \mid \varnothing \vdash t : A$ as $t : A$.

▷ Claim 2.1 (Affineness). If $\lambda x.\, t$ is well-typed, the variable x occurs at most once in t, at depth 0. (But let-bound variables are not subject to any such restriction).

Normalization. Our β-reduction rules, which can be applied in any context C, are as follows, where the context L consists of a succession of let-binders (i.e. $L ::= \langle\cdot\rangle \mid \texttt{let}\ !x' = t'\ \texttt{in}\ L$):

$$L\langle\lambda x.\, t\rangle\ u \longrightarrow_\beta L\langle t\{x := u\}\rangle \qquad \texttt{let}\ !x = L\langle !u\rangle\ \texttt{in}\ t \longrightarrow_\beta L\langle t\{x := u\}\rangle$$

For instance, the following is a valid β-reduction:

$$(\texttt{let}\ !x = u\ \texttt{in}\ \texttt{let}\ !y = v\ \texttt{in}\ \lambda z.\, z\ x\ y)\ t \quad \longrightarrow_\beta^* \quad \texttt{let}\ !x = u\ \texttt{in}\ \texttt{let}\ !y = v\ \texttt{in}\ t\ x\ y$$

This "reduction at a distance" – an idea of Accattoli & Kesner [3] – is a way to get the desirable Proposition 2.4 below without having to introduce cumbersome "commuting conversions". For an extended discussion in the context of a system very close to ours, see [32, §1.2.1].

▶ **Proposition 2.2** (Normalization and subject reduction). *Any well-typed term has a β-normal form. Furthermore, if $\Theta \mid \Phi \vdash t : A$ then $\Theta \mid \Phi \vdash t' : A$ for any β-normal form t' of t.*

▶ **Definition 2.3.** *We say that a term $\Theta \mid \Phi \vdash t : A$ is purely affine when* all of its subterms *have purely affine types (cf. Definition 1.3) and $\Theta = \emptyset$, which implies that it contains no !-box or let-binding. We also call almost purely affine (resp. almost !-depth 1) the terms $\Theta \mid \Phi \vdash t : A$ in which, for every subterm u of t,*

- *the type of u is almost purely affine (resp. almost !-depth 1 – cf. Definition 1.6),*
- *if $u = !r$ then r is purely (resp. almost purely) affine,*
- *$\Theta = x_1 : o, \dots, x_n : o$ (resp. $\Theta = x_1 : A_1, \dots, x_n : A_n$ where $A_1, \dots, A_n$ are almost purely affine).*

▶ **Proposition 2.4.** *Assume that we work with purely affine constants, as will always be the case in this paper. Let $\Theta \mid \Phi \vdash t : A$ and* suppose t is in normal form. *If A and the types in Θ and Φ are purely affine (resp. almost purely affine, almost !-depth 1), then so is t.*

For example, $\texttt{let}\ !z = !(\lambda x.\ x)\ \texttt{in}\ z : o \multimap o$ is not purely affine but its normal form $\lambda x.\ x$ is.

Encoding of trees in our affine λ-calculus. Fix a ranked alphabet Σ. We consider λ-terms built over the constants $c : o^{\mathrm{rk}(c)} \multimap o$ for $c \in \Sigma$. There is a canonical encoding $\widetilde{(\cdot)}$ of trees as closed terms of type o; for instance, $\tau = a(b(c), c)$ is encoded as $\widetilde{\tau} = a\ (b\ c)\ c$.

▶ **Proposition 2.5.** *Every closed term* of type o *using these constants admits a* unique *normal form. Furthermore, $\widetilde{(\cdot)}$ is a* bijection *between* $\mathrm{Tree}(\Sigma)$ *and these normal forms.*

Given a type A and a family of λ-terms $\vec{t} = (t_c)_{c \in \Sigma}$ such that $t_c : A^{\mathrm{rk}(c)} \multimap A$ for each letter $c \in \Sigma$, we write $\widehat{\tau}(\vec{t})$ for the result of replacing each constant c in $\widetilde{\tau}$ by t_c. It is always well typed, with type A. For the example $\tau = a(b(c), c)$, we have $\widehat{\tau}((t_x)_{x \in \{a,b,c\}}) = t_a\ (t_b\ t_c)\ t_c$.

Higher-order transducers (or λ-transducers). Let us fix an input ranked alphabet Γ.

▶ **Definition 2.6.** *An* (affine) λ-transducer $\mathrm{Tree}(\Gamma) \to \mathrm{Tree}(\Sigma)$ *is specified by a* memory type A *and a family of terms (that can use the aforementioned constants from Σ):*

$$\underbrace{t_a : A^{\mathrm{rk}(a)} \multimap A}_{\text{“transition terms”}} \text{ for each letter } a \in \Gamma \qquad \text{and} \qquad \underbrace{u : A \multimap o}_{\text{“output term”}}$$

The λ-transducer defines the function

$$\tau \in \mathrm{Tree}(\Gamma) \quad \mapsto \quad \underbrace{\sigma \in \mathrm{Tree}(\Sigma) \text{ such that } \widetilde{\sigma} \text{ is the normal form of } \overbrace{u\ \widehat{\tau}((t_a)_{a \in \Gamma})}^{\text{well-typed with type } o}}_{\text{well-defined and unique thanks to Proposition 2.5}}$$

This amounts to specifying a structurally recursive function over $\mathrm{Tree}(\Gamma)$ with return type A, followed by some post-processing that produces an output tree. Alternatively, a λ-transducer can be seen as a kind of tree automaton whose memory consists of affine λ-terms of some type A (with constants from Σ) and whose bottom-up transitions are also defined by λ-terms.

In addition to the purely affine Example 1.1, we exhibit two other λ-transducers.

▶ **Example 2.7.** The following almost affine λ-transducer maps $S^n(0) = S(\dots(S(0)))$ to the list $[1, 2, \dots, n]$, encoded as the tree $\texttt{cons}(S(0), \dots(\texttt{cons}(S^n(0), \texttt{nil})\dots)$.

$t_0 = \lambda x.\ \texttt{nil} : !o \multimap o$ (memory type) $\qquad u = \lambda g.\ g\ !(S\ 0)$

$t_S = \lambda g.\ \lambda x.\ \texttt{let}\ !y = x\ \texttt{in}\ \texttt{cons}\ y\ (g\ !(S\ y))$

▶ **Example 2.8.** The following λ-transducer takes as input the binary encoding of a natural number n and returns a complete binary tree of height n, e.g. $\mathtt{0}(\mathtt{0}(\mathtt{1}(\mathtt{0}(\varepsilon)))) \mapsto a(a(c,c),a(c,c))$. Thus, its growth is doubly exponential. Its memory type $!(!o \multimap !o) \multimap o$ is almost !-depth 1.

$$t_0 = \lambda g.\ \lambda x.\ \mathtt{let}\ !f = x\ \mathtt{in}\ g\ !(\lambda y.\ f\ (f\ y)) \qquad t_\varepsilon = \lambda x.\ \mathtt{let}\ !f = x\ \mathtt{in}\ \mathtt{let}\ !z = f\ !c\ \mathtt{in}\ z$$
$$t_1 = \lambda g.\ \lambda x.\ \mathtt{let}\ !f = x\ \mathtt{in}\ g\ !(\lambda y.\ \mathtt{let}\ !z = f\ (f\ y)\ \mathtt{in}\ !(a\ z\ z)) \qquad u = \lambda g.\ g\ !(\lambda y.\ y)$$

Moreover, composing Examples 1.1 and 2.7 according to Proposition 1.8 gives another almost purely affine example.

▶ Remark 2.9. The original impetus for Engelfriet and Vogler's higher-order transducers [21] was that their transducers of order k are equivalent to compositions of k unrestricted macro tree transducers. A major motivation of Gallot et al.'s machine model using linear λ-terms was also function composition, for which they give efficient constructions [23, Chapter 6] (which are non-trivial). And in Nguyễn and Pradic's "implicit automata", composition is just a matter of plugging two λ-terms together [37, Lemma 2.8].

3 Tree-walking transducers (last definitions needed for Theorem 1.4)

Generalities. In this paper, we shall encounter several machine models that generate some output tree in a top-down fashion, starting from the root. (This is not the case of λ-transducers.) They follow a common pattern, which we abstract as a lightweight formalism here: essentially, a deterministic regular tree grammar with infinitely many non-terminals.

▶ Remark 3.1. Engelfriet's tree grammars with storage (see for instance [17]) are more complex formalisms that also attempt to unify several definitions of tree transducer models.

▶ **Definition 3.2.** *A* tree-generating machine *over the ranked alphabet* Σ *consists of:*

- *a (possibly infinite) set* $\mathcal{K}$ *of* configurations*;*
- *an* initial configuration $\kappa_0 \in \mathcal{K}$ *– in concrete instances,* κ_0 *will be defined as a simple function of some input tree (for tree transducers) or some given* λ*-term (for the IAM);*
- *a* computation-step (partial) function $\mathcal{K} \rightharpoonup \mathrm{Tree}(\Sigma, \mathcal{K})$.

▶ **Example 3.3.** To motivate the formal semantics for these machines that we will soon define, we give a tree-generating machine that is meant to produce the list $[1, 2, \dots, n]$ (for an arbitrarily chosen $n \in \mathbb{N}$), encoded as in Example 2.7.

- The set of configurations is $\mathcal{K} = \{\mathsf{spine}, \mathsf{num}\} \times \mathbb{N}$ where spine and num are formal symbols.
- The initial configuration is (spine, n) – let us write this pair as $\langle \mathsf{spine}, n \rangle$.
- The computation-step function is
$$\begin{cases} \langle \mathsf{spine}, 0 \rangle \mapsto \mathtt{nil} \\ \langle \mathsf{spine}, m+1 \rangle \mapsto \mathtt{cons}(\langle \mathsf{num}, n-m \rangle, \langle \mathsf{spine}, m \rangle) \\ \langle \mathsf{num}, 0 \rangle \mapsto 0 \\ \langle \mathsf{num}, m+1 \rangle \mapsto S(\langle \mathsf{num}, m \rangle) \end{cases}$$

For $n = 3$, one possible run is

$$\langle \mathsf{spine}, 3 \rangle \rightsquigarrow \mathtt{cons}(\langle \mathsf{num}, 1 \rangle, \langle \mathsf{spine}, 2 \rangle) \rightsquigarrow \mathtt{cons}(S(\langle \mathsf{num}, 0 \rangle), \langle \mathsf{spine}, 2 \rangle) \rightsquigarrow \dots$$

All runs starting from $\langle \mathsf{spine}, 3 \rangle$ eventually reach the tree that encodes $[1, 2, 3]$.

Let us now discuss the general case. Intuitively, the execution of the machine involves spawning several independent concurrent processes, outputting disjoint subtrees. We formalize this parallel computation as a rewriting system $\rightsquigarrow$ on $\mathrm{Tree}(\Sigma, \mathcal{K})$: we have $\tau_1 \rightsquigarrow \tau_2$ whenever

τ_2 is obtained from τ_1 by substituting one of its configuration leaves by its image by the computation-step function. This rewriting system is orthogonal, and therefore confluent, which means that the initial configuration has at most one normal form. If this normal form exists and belongs to $\mathrm{Tree}(\Sigma)$, it is the output of the machine; we then say that the machine converges. Otherwise, the output is undefined; the machine diverges.

Tree-walking transducers. Before giving the definition, let us see a concrete example.

▶ **Example 3.4.** According to Theorem 1.4, since the λ-transducer of Example 1.1 is purely affine, the function "count non-a nodes" that it defines can also be computed by some (reversible) tree-walking transducer. We show the run of such a transducer on the input $a_1(b_2(c_3), c_4)$ – the indices are not part of the node labels, they serve to distinguish positions:

$$\begin{aligned}(q, \circlearrowleft, a_1) &\rightsquigarrow (q, \downarrow_\bullet, b_2) \rightsquigarrow S((q, \downarrow_\bullet, c_3)) \rightsquigarrow S(S((q, \uparrow_1^\bullet, b_2))) \rightsquigarrow S(S((q, \uparrow_1^\bullet, a_1)))\\ &\rightsquigarrow S(S((q, \downarrow_\bullet, c_4))) \rightsquigarrow S(S(S((q, \uparrow_2^\bullet, a_1))))) \rightsquigarrow S(S(S(0)))\end{aligned}$$

where q is the single state of the transducer. The second component records the "provenance", i.e. the previous position of the tree-walking transducer relatively to the current node (stored in the third component): $\downarrow_\bullet$ refers to its parent, $\circlearrowleft$ to itself, and $\uparrow_i^\bullet$ to its i-th child.

▶ **Definition 3.5.** *A* tree-walking transducer (TWT) $\mathrm{Tree}(\Gamma) \multimap \mathrm{Tree}(\Sigma)$ *consists of:*

- *a finite set of* states Q *with an* initial state $q_0 \in Q$
- *a family of (partial)* transition functions *for* $a \in \Gamma$

$$\delta_a \colon Q \times \{\downarrow_\bullet, \circlearrowleft, \uparrow_1^\bullet, \dots, \uparrow_k^\bullet\} \multimap \mathrm{Tree}(\Sigma,\ Q \times \{\uparrow_\bullet, \circlearrowleft, \downarrow_1^\bullet, \dots, \downarrow_k^\bullet\}) \quad \textit{where } k = \mathrm{rk}(a)$$

- *a family of (partial)* transition functions at the root *for* $a \in \Gamma$

$$\delta_a^{\mathrm{root}} \colon Q \times \{\circlearrowleft, \uparrow_1^\bullet, \dots, \uparrow_k^\bullet\} \multimap \mathrm{Tree}(\Sigma,\ Q \times \{\circlearrowleft, \downarrow_1^\bullet, \dots, \downarrow_k^\bullet\}) \quad \textit{where } k = \mathrm{rk}(a)$$

The TWT associates to each input tree τ a tree-generating machine whose output is the image of τ. Its set of configurations is $Q \times \{\downarrow_\bullet, \circlearrowleft, \uparrow_1^\bullet, \dots\} \times \{\text{nodes of } \tau\}$ and its initial configuration of is $(q_0,\ \circlearrowleft,\ \text{root of } \tau)$.

To define the image of (q, p, v) by the computation-step function (it is undefined if one of the following steps is undefined), we start with either $\delta_a^{\mathrm{root}}(q, p)$ if v is the root or $\delta_a(q, p)$ otherwise – where a is the label of the node v – then replace each $(q', p) \in Q \times \{\uparrow_\bullet, \dots\}$ by

$$\begin{cases} (q',\ \downarrow_\bullet,\ i\text{-th child of } v) & \text{if } p = \downarrow_i^\bullet \\ (q',\ \uparrow_j^\bullet,\ \text{parent of } v) & \text{if } p = \uparrow_\bullet \text{ and } v \text{ is the } j\text{-th child of its parent} \end{cases} \qquad (q', \circlearrowleft, v) \quad \text{if } p = \circlearrowleft$$

▶ **Example 3.6.** Using the idea of Example 3.3, we define a tree-walking transducer that computes the function "$n \mapsto [1, \dots, n]$ modulo encodings" of Example 2.7. Its set of states is $Q = \{\mathsf{spine}, \mathsf{num}\}$, its initial state is spine and its transitions are

$$\begin{aligned}\delta_S^{\mathrm{root}}(\mathsf{spine}, \circlearrowleft) = \delta_S(\mathsf{spine}, \downarrow_\bullet) &= \mathtt{cons}((\mathsf{num}, \circlearrowleft), (\mathsf{spine}, \downarrow_1^\bullet))\\ \delta_S(\mathsf{num}, \circlearrowleft) = \delta_S(\mathsf{num}, \uparrow_1^\bullet) &= S((\mathsf{num}, \uparrow_\bullet))\\ \delta_S^{\mathrm{root}}(\mathsf{num}, \circlearrowleft) = \delta_S^{\mathrm{root}}(\mathsf{num}, \uparrow_1^\bullet) &= S(0) \qquad \delta_0^{\mathrm{root}}(\mathsf{spine}, \circlearrowleft) = \delta_0(\mathsf{spine}, \downarrow_\bullet) = \mathtt{nil}\end{aligned}$$

▶ Remark 3.7. In the above example, the provenance information in $\{\downarrow_\bullet, \dots\}$ plays no role. On the contrary, it is crucial in the former Example 3.4, where we have $\delta_a^{\mathrm{root}}(q, \uparrow_2^\bullet) = 0$ and

$$\delta_a^{\mathrm{root}}(q, \circlearrowleft) = \delta_a(q, \downarrow_\bullet) = (q, \downarrow_1^\bullet) \qquad \underbrace{\delta_a^{\mathrm{root}}(q, \uparrow_1^\bullet) = \delta_a(q, \uparrow_1^\bullet) = (q, \downarrow_2^\bullet)}_{\text{"after returning from the 1st child, the traversal starts visiting the 2nd child"}} \qquad \delta_a(q, \uparrow_2^\bullet) = (q, \uparrow_\bullet)$$

In the definitions of tree-walking automata or transducers in the literature, e.g. [9, 16, 19], transitions often do not have access to this provenance, but instead they can depend on the "child number" i of the current node (such that the node is an i-th child of its parent). One can easily simulate one variant with the other; but if neither of these features were available, the machine model would be strictly weaker [25, Theorem 5.5].

Our main motivation for using provenances instead of child numbers is that, according to [39, Sections 5 and 6], being "direction-determinate" – i.e. knowing which previous node the current configuration came from – is important in the reversible case. This can indeed be observed in the proof of Claim 3.9. (Directed states are used in [15] for a similar reason.)

Let us say that a map $\delta\colon X \to \mathrm{Tree}(\Sigma, Y)$ is *Y-leaf-injective* whenever in the family of trees $(\delta(x) \mid x \in X)$, each $y \in Y$ occurs at most once: y appears in at most one tree of the family, and if it does appear, this tree has a single leaf with label y.

▶ **Definition 3.8.** *A tree-walking transducer* $\mathrm{Tree}(\Gamma) \rightharpoonup \mathrm{Tree}(\Sigma)$ *is* reversible *when all maps* δ_a *and* δ_a^{root} *for* $a \in \Gamma$ *are* $(Q \times \{\uparrow_\bullet, \circlearrowleft, \downarrow_1^\bullet, \dots, \downarrow_{\mathrm{rk}(a)}^\bullet\})$*-leaf-injective.*

Example 3.4 is reversible, but not Example 3.6.

▷ Claim 3.9 (The meaning of reversibility). Fix a reversible tree-walking transducer and an input tree. Any configuration C has at most one predecessor configuration, i.e. one configuration C' whose image by the computation-step function contains C as a leaf.

Proof. Let $(q, p, v) = C$, and assume that $(q', p', v') = C'$ exists. The provenance information p tells us where v' is situated in the input tree relatively to v, so we determine v' along with its label a. Among the leaves of $\delta_a(q', p')$ (or $\delta_a^{\mathrm{root}}(q', p')$ if v' is the root), there must be one of the form (q, p) where q and p are related (in a one-to-one fashion) as defined just above Example 3.6. By leaf-injectivity this uniquely determines q' and p'. ◁

4 From the Interaction Abstract Machine to (reversible) TWTs

This section finally discusses the proof of Theorem 1.4. (For lack of space, we focus on stating some key definitions and illustrating the ideas on examples; rigorous details are left to the technical report.) By definition, computing the image of an input tree by a λ-transducer involves normalizing a λ-term. Unfortunately, iterating β-reductions until a normal form is reached requires too much working memory to be implemented by a finite-state device, such as a tree-walking transducer. That is why we rely on other ways to normalize terms of base type o, namely variants of the Interaction Abstract Machine (IAM).

The purely affine IAM. Let us start with a machine that can normalize a purely affine term $v : o$. Intuitively, it moves a token around the *edges* of the syntax tree of v; we represent the situation where the token is on the edge connecting the subterm t to its context C (such that $C\langle t\rangle = v$) and is moving down (resp. up) as $C\langle \underline{t}\rangle$ (resp. $C\langle \overline{t}\rangle$). The token carries a small amount of additional information: a "tape" which is a stack using the symbols $\bullet$ and $\circ$.

We reuse the formalism of tree-generating machines from the previous section, and we denote by X^* (with the Kleene star) the set of lists with elements in X.

▶ **Definition 4.1.** *Let $v : o$ be purely affine. The tree-generating machine* $\mathrm{PAIAM}(v)$ *has:*

- *configurations of the form (d, t, C, T) where $d \in \{\downarrow, \uparrow\}$, $C\langle t\rangle = v$ and $T \in \{\bullet, \circ\}^*$ – which we abbreviate as $(C\langle \underline{t}\rangle, T)$ when $d = \downarrow$ or $(C\langle \overline{t}\rangle, T)$ when $d = \uparrow$;*
- *the initial configuration $(\underline{t}, \varepsilon) = (\downarrow, t, \langle\cdot\rangle, \varepsilon)$;*

- *the following computation-step function:*

$$\begin{array}{rcl@{\qquad}rcl}
(C\langle \underline{t\ u}\rangle, T) & \mapsto & (C\langle \underline{t}\ u\rangle, \bullet\cdot T) & (C\langle \overline{t}\ u\rangle, \bullet\cdot T) & \mapsto & (C\langle \overline{t\ u}\rangle, T)\\
(C\langle t\ \overline{u}\rangle, T) & \mapsto & (C\langle \underline{t}\ u\rangle, \circ\cdot T) & (C\langle \overline{t}\ u\rangle, \circ\cdot T) & \mapsto & (C\langle t\ \underline{u}\rangle, T)\\
(C\langle \lambda x.\ \overline{t}\rangle, T) & \mapsto & (C\langle \overline{\lambda x.\ t}\rangle, \bullet\cdot T) & (C\langle \underline{\lambda x.\ t}\rangle, \bullet\cdot T) & \mapsto & (C\langle \lambda x.\ \underline{t}\rangle, T)\\
(C\langle \lambda x.\ D\langle \underline{x}\rangle\rangle, T) & \mapsto & (C\langle \overline{\lambda x.\ D\langle x\rangle}\rangle, \circ\cdot T) & (C\langle \underline{\lambda x.\ D\langle x\rangle}\rangle, \circ\cdot T) & \mapsto & (C\langle \lambda x.\ D\langle \overline{x}\rangle\rangle, T)\\
(C\langle \underline{c}\rangle, \bullet^{\mathrm{rk}(c)}\cdot T) & \mapsto & c\Big((C\langle \overline{c}\rangle, \circ\cdot T), (C\langle \overline{c}\rangle, \bullet\cdot\circ\cdot T), \dots, (C\langle \overline{c}\rangle, \bullet^{\mathrm{rk}(c)-1}\cdot\circ\cdot T)\Big) & & &
\end{array}$$

The last rule handles constants $c : o^{\mathrm{rk}(c)} \multimap o$ coming from a fixed ranked alphabet Σ. When $\mathrm{rk}(c) = 0$, the right-hand side is simply the tree with a single node labeled by c; otherwise, it is a tree of height 1, whose leaves are all configurations.

▶ **Example 4.2.** Let u and $(t_x)_{x\in\{a,b,c\}}$ be the terms defining the purely affine λ-transducer from Example 1.1. On the term $v = u\ (t_a\ (t_b\ t_c)\ t_c)$, we have the following IAM execution (where $C \rightsquigarrow^n C'$ means that C rewrites into C' in n steps):

$$\begin{array}{rl}
(\underline{v}, \varepsilon) \rightsquigarrow & (\underline{u}\ (t_a\ (t_b\ t_c)\ t_c), \bullet) \rightsquigarrow ((\lambda f.\ \underline{f\ 0})\ (t_a\ (t_b\ t_c)\ t_c), \varepsilon) \rightsquigarrow ((\lambda f.\ \underline{f}\ 0)\ (\dots), \bullet)\\
\rightsquigarrow & (\overline{(\lambda f.\ f\ 0)}\ (t_a\ (t_b\ t_c)\ t_c), \circ\bullet) \rightsquigarrow (u\ \underline{(t_a\ (t_b\ t_c)\ t_c)}, \bullet) \rightsquigarrow^2 (u\ (\underline{t_a}\ (t_b\ t_c)\ t_c), \bullet\bullet\bullet)\\
\rightsquigarrow^4 & (u\ ((\lambda\ell.\ \lambda r.\ \lambda x.\ \underline{\ell}\ (r\ x))\ (t_b\ t_c)\ t_c), \bullet) \rightsquigarrow (u\ (\overline{t_a}\ (t_b\ t_c)\ t_c), \circ\bullet) \rightsquigarrow^2 (\dots \underline{t_b} \dots, \bullet\bullet)\\
\rightsquigarrow^3 & (u\ (t_a\ ((\lambda f.\ \lambda x.\ \underline{S}\ (f\ x))\ t_c)\ t_c), \bullet) \rightsquigarrow \underline{S}((u\ (t_a\ ((\lambda f.\ \lambda x.\ \overline{S}\ (f\ x))\ t_c)\ t_c), \circ))\\
\rightsquigarrow^* & \dots [\text{many steps}] \dots \qquad \text{output node (will be the root of the output tree)}\\
\rightsquigarrow^* & S(S((u\ (\dots \underline{t_c}), \bullet))) \rightsquigarrow S^3((u\ (t_a\ (t_b\ t_c)\ \overline{S}), \circ)) \rightsquigarrow S^3((u\ \underline{(t_a\ (t_b\ t_c)}\ S), \circ\circ))\\
\rightsquigarrow^2 & S^3((u\ ((\lambda\ell.\ \underline{\lambda r.\ \lambda x.\ \ell\ (r\ x)})\ (t_b\ t_c)\ t_c), \circ\circ)) \rightsquigarrow S^3((\dots (\overline{r}\ x) \dots, \circ))\\
\rightsquigarrow & S^3((\dots (r\ \underline{x}) \dots, \varepsilon)) \rightsquigarrow S^3((u\ (\overline{(\lambda\ell.\ \lambda r.\ \lambda x.\ \ell\ (r\ x))}\ (t_b\ t_c)\ t_c), \circ))\\
\rightsquigarrow^2 & S^3((u\ (\overline{t_a}\ (t_b\ t_c)\ t_c), \bullet\bullet\circ)) \rightsquigarrow^2 S^3((u\ \overline{(t_a\ (t_b\ t_c)\ t_c)}, \circ)) \rightsquigarrow S^3((\underline{u} \dots, \circ\circ))\\
\rightsquigarrow & S^3(((\lambda f.\ \overline{f}\ 0) \dots, \circ)) \rightsquigarrow S^3(((\lambda f.\ f\ \underline{0}) \dots, \varepsilon)) \rightsquigarrow S(S(S(0)))
\end{array}$$

Aside from our bespoke extension dedicated to constants from Σ, all the other rules in Definition 4.1 come from the "Linear IAM" described by Accattoli et al. [1, Section 3] (see also [46, §3.1]) – we refer to those papers for high-level explanations of these rules. Despite its name, the Linear IAM also works for affine terms (cf. [32, §3.3.4]). In particular, when run on closed normal forms of type o, namely encoding of trees by Prop. 2.5, the IAM outputs the encoded tree.

▶ **Proposition 4.3.** *For each tree $\tau \in \mathrm{Tree}(\Sigma)$, $(\widetilde{\tau}, \varepsilon) \rightsquigarrow^* \tau$.*

Proof. We strengthen the statement, considering the reduction $(C\langle\widetilde{\tau}\rangle, \varepsilon) \rightsquigarrow^* \tau$. Then, we proceed by induction on the structure of τ. If τ is a leaf, then we have $\widetilde{\tau} = c$ and $\mathrm{rk}(c) = 0$. Thus, we have $(C\langle\underline{c}\rangle, \varepsilon) \rightsquigarrow c$, which concludes this case. Otherwise τ is not a leaf, and in particular it is of the form $c\ (\tau_1 \dots \tau_k)$, where $k = \mathrm{rk}(c) \geq 1$. Then we have:

$$\begin{array}{rcl}
(C\langle \underline{c\ \widetilde{\tau}_1 \dots \widetilde{\tau}_k}\rangle, \varepsilon) & \rightsquigarrow^k & (C\langle \underline{c}\ \widetilde{\tau}_1 \dots \widetilde{\tau}_k\rangle, \bullet^k)\\
& \rightsquigarrow & c\ \Big((C\langle \overline{c}\ \widetilde{\tau}_1 \dots \widetilde{\tau}_k\rangle), \circ, \dots, (C\langle \overline{c}\ \widetilde{\tau}_1 \dots \widetilde{\tau}_k\rangle, \bullet^{k-1}\cdot\circ)\Big)\\
& \rightsquigarrow^* & c\ \Big((C\langle c\ \underline{\widetilde{\tau}_1}\widetilde{\tau}_2 \dots \widetilde{\tau}_k\rangle, \varepsilon), \dots, (C\langle c\ \widetilde{\tau}_1 \dots \widetilde{\tau}_{k-1}\underline{\widetilde{\tau}_k}\rangle, \varepsilon)\Big)\\
& \rightsquigarrow^* & c\ (\tau_1 \dots \tau_k)
\end{array}$$

Where the last reduction comes from applying the induction hypothesis to each configuration. ◀

The soundness of the machine comes from the standard fact (see e.g. [1, 31]) that what is computed by the IAM, in our case the tree, is invariant by β-reduction, the details are in the technical report [38].

▶ **Theorem 4.4** (Soundness of the purely affine IAM). *For any purely affine term $v : o$, the output of* PAIAM(v) *is the unique $\tau \in \mathrm{Tree}(\Sigma)$ such that $\widetilde{\tau}$ is the normal form of v.*

Simulating λ-transducers by tree-walking transducers. Fix a purely affine λ-transducer given by $t_a : A^{\mathrm{rk}(a)} \multimap A$ for $a \in \Gamma$ and $u : A \multimap o$. Thanks to Proposition 2.2, we may assume that u and every t_a are in normal form. Proposition 2.4 then tells us that since A is purely affine, so are these terms – and therefore, so is $u\,\widehat{\tau}((t_a)_{a\in\Gamma}) : o$ for any input $\tau \in \mathrm{Tree}(\Gamma)$. Thus, we can use the purely affine IAM to compute the tree encoded by its normal form – which, by definition, gives us the image of τ by our λ-transducer.

To prove the first part of Theorem 1.4, we just need to simulate (in an appropriate sense, made precise in the technical report [38]) PAIAM$(u\,\widehat{\tau}((t_a)_{a\in\Gamma}))$ by a TWT running on τ. Example 4.5 below illustrates how this works: a configuration of PAIAM$(u\,\widehat{\tau}((t_a)_{a\in\Gamma}))$ is represented by a configuration on input τ of a TWT in which each state consists of

- a formal symbol in $\{\mathsf{U}, \mathsf{T}, \nabla, \Delta\}$;
- if the symbol is U (resp. T), a position in u (resp. $t_a\ \Diamond_1\ \dots\ \Diamond_{\mathrm{rk}(a)}$ for some $a \in \Gamma$);
- a tape in $\{\bullet, \circ\}^*$ that appears in the run of PAIAM$(u\,\widehat{\sigma}((t_a)_{a\in\Gamma}))$ for some $\sigma \in \mathrm{Tree}(\Gamma)$.

The information in the first two components is clearly bounded, but this is not so obvious for the third one. We shall address this issue using Corollary 4.7 below, after the example.

▶ **Example 4.5.** We translate Example 1.1 to a TWT that has the following run on the input $a_1(b_2(c_3), c_4)$ – note how the steps correspond to those of the IAM run in Example 4.2. This run visits the same input nodes as Example 3.4, in the same order. The only difference is that it stays for longer on each node ($\circlearrowleft$ appears very frequently).

$$
\begin{aligned}
(\mathsf{I}, \circlearrowleft, a_1) &\rightsquigarrow (\mathsf{U}(\underline{\lambda f.\ f\ 0}, \bullet), \circlearrowleft, a_1) \rightsquigarrow^3 (\mathsf{U}(\overline{\lambda f.\ f\ 0}, \circ\bullet), \circlearrowleft, a_1) \rightsquigarrow (\nabla(\bullet), \circlearrowleft, a_1) \\
&\rightsquigarrow (\mathsf{T}(\underline{t_a\ \Diamond_1}\ \Diamond_2, \bullet\bullet), \circlearrowleft, a_1) \rightsquigarrow^5 (\mathsf{T}((\lambda\ell.\ \lambda r.\ \lambda x.\ \underline{\ell}\ (r\ x))\ \Diamond_1\ \Diamond_2, \bullet), \circlearrowleft, a_1) \\
&\rightsquigarrow (\mathsf{T}(\overline{t_a}\ \Diamond_1\ \Diamond_2, \circ\bullet), \circlearrowleft, a_1) \rightsquigarrow (\nabla(\bullet), \downarrow_\bullet, b_2) \rightsquigarrow (\mathsf{T}(\underline{t_b}\ \Diamond_1, \bullet\bullet), \circlearrowleft, b_2) \\
&\rightsquigarrow^3 (\mathsf{T}((\lambda f.\ \lambda x.\ \underline{S}\ (f\ x))\ \Diamond_1, \bullet), \circlearrowleft, b_2) \rightsquigarrow S((\mathsf{T}((\lambda f.\ \lambda x.\ \overline{S}\ (f\ x))\ \Diamond_1, \circ), \circlearrowleft, b_2)) \\
&\rightsquigarrow^* \dots [\text{many steps}] \dots \\
&\rightsquigarrow^* S(S((\nabla(\bullet), \downarrow_\bullet, c_4))) \rightsquigarrow S^3((\Delta(\circ), \uparrow_2^\bullet, a_1)) \rightsquigarrow (\mathsf{T}(\underline{t_a\ \Diamond_1}\ \Diamond_2, \circ\circ), \circlearrowleft, a_1) \\
&\rightsquigarrow^7 S^3(\mathsf{T}(\overline{t_a}\ \Diamond_1\ \Diamond_2), \bullet\bullet\circ), \circlearrowleft, a_1) \rightsquigarrow^2 S^3((\Delta(\circ), \circlearrowleft, a_1)) \rightsquigarrow S^3(\mathsf{U}(\underline{u}, \circ\circ), \circlearrowleft, a_1) \\
&\rightsquigarrow^2 S^3((\mathsf{U}(\lambda f.\ f\ \underline{0}, \varepsilon), \circlearrowleft, a_1)) \rightsquigarrow S(S(S(0)))
\end{aligned}
$$

Finite states via a typing invariant. To bound the size of tapes $T \in \{\bullet, \circ\}^*$, we leverage the type system. The idea is that a tape that appears in an IAM run "points to" an occurrence of the base type o in the type of the current subterm. Formally, we define inductively:

$$A \lightning \varepsilon = A \qquad (A \multimap B) \lightning (\circ \cdot T) = A \lightning T \qquad (A \multimap B) \lightning (\bullet \cdot T) = B \lightning T$$

(thus, $o \lightning T$ is undefined for $T \neq \varepsilon$). We then have the following invariant on configurations:

▶ **Proposition 4.6** (compare with [32, Lemma 32 in §3.3.5]). *Suppose that either $(C\langle \underline{t} \rangle, T)$ or $(C\langle \overline{t} \rangle, T)$ appears in a run of* PAIAM(v) *for some $v : o$. If the (not necessarily closed) term t is given the type A as part of a typing derivation for $v : o$, then $A \lightning T = o$.*

Since $|T| \leq \mathrm{height}(A)$ is a necessary condition for $A \lightning T$ to be defined, we immediately get:

▶ **Corollary 4.7.** *The sizes of the tapes that appear in a run of* PAIAM(v) *are bounded by the maximum, over all subterms t of v, of the height of the syntax tree of the type of v.*

Therefore, the tapes that can appear in a run of PAIAM($u\,\widehat{\tau}((t_a)_{a\in\Gamma})$) have their size bounded depending only on the subterms of u and of each t_a, independently of $\tau \in \mathrm{Tree}(\Gamma)$. This ensures that the TWT we build from a purely affine λ-transducer has a finite set of states – which concludes our exposition of the key ingredients for the first half of Theorem 1.4.

The almost purely affine case. To prove the second half of Theorem 1.4, we add non-standard rules for let-bindings and !-boxes to the Interaction Abstract Machine.

▶ **Definition 4.8.** *Let $v : o$ be an almost purely affine term.* APAIAM(v) *is the extension of* PAIAM(v) *(cf. Definition 4.1) with the following new cases in the computation-step function:*

$$(C\langle \underline{\texttt{let } !x = u \texttt{ in } t}\rangle, T) \mapsto (C\langle \texttt{let } !x = u \texttt{ in } \underline{t}\rangle, T) \qquad (C\langle \underline{!t}\rangle, T) \mapsto (C\langle !\underline{t}\rangle, T)$$
$$(C\langle \texttt{let } !x = u \texttt{ in } \overline{t}\rangle, T) \mapsto (C\langle \overline{\texttt{let } !x = u \texttt{ in } t}\rangle, T)$$
$$(C\langle \texttt{let } !x = u \texttt{ in } D\langle \underline{x}\rangle\rangle, T) \mapsto (C\langle \texttt{let } !x = \underline{u} \texttt{ in } D\langle x\rangle\rangle, T)$$

▷ Claim 4.9. Proposition 4.6 extends to APAIAM(v) with $!A \lightning T = A \lightning T$.

The last rule (and the rule for !-boxes) in Definition 4.8 break(s) a key duality principle at work in the purely affine IAM (and suggested by the layout of Definition 4.1): if any rule – except the one for constants from Σ – sends a configuration κ_1 to another configuration κ_2, then there is a dual rule sending $\kappa_2^\perp$ to $\kappa_1^\perp$, where $(C\langle \underline{t}\rangle, T)^\perp = (C\langle \overline{t}\rangle, T)$ and conversely $(C\langle \overline{t}\rangle, T)^\perp = (C\langle \underline{t}\rangle, T)$.

In fact, our new rule for let-bound variables $\underline{x}$ *cannot* have a dual, because it is not injective. Indeed, consider a term of the form $C\langle \texttt{let } !x = u \texttt{ in } t\rangle$ where t contains multiple occurrences of x, i.e. $t = D_1\langle x\rangle = D_2\langle x\rangle$ for some contexts $D_1 \neq D_2$ – this may happen, since let-bound variables are not affine. Then for any $T \in \{\bullet, \circ\}^*$, the computation-step function sends both $(C\langle \texttt{let } !x = u \texttt{ in } D_1\langle \underline{x}\rangle\rangle, T)$ and $(C\langle \texttt{let } !x = u \texttt{ in } D_2\langle \underline{x}\rangle\rangle, T)$ to the same configuration $(C\langle \texttt{let } !x = \underline{u} \texttt{ in } t\rangle, T)$ due to the $\underline{x}$ rule. This is why *reversible* TWTs can simulate the purely affine IAM, but not the *almost* purely affine IAM.

To be sure that the missing dual rule is unnecessary, we show that configurations of the form $(C\langle \texttt{let } !x = \overline{u} \texttt{ in } t\rangle, T)$ cannot occur in an actual run. Assume the opposite for the sake of contradiction. The typing rule for let-bindings forces the type of u to have the form $!A$, and by almost pure affineness, it must be $!o$. By Claim 4.9, $!o \lightning T = o \lightning T$; therefore, $T = \varepsilon$, which contradicts another invariant:

▶ **Proposition 4.10.** *If $(C\langle \overline{t}\rangle, T)$ (resp. $(C\langle \underline{t}\rangle, T)$) appears in a run of* APAIAM(v) *for some almost purely affine $v : o$, then T contains an odd (resp. even) number of $\circ$s.*

(The same reasoning also shows that $(C\langle !\overline{t}\rangle, T)$ cannot occur in a run).

Having ruled out these problematic configurations, we can establish soundness for the almost purely affine IAM exactly as before, extending Theorem 4.4.

▶ **Proposition 4.11** (Soundness of the almost purely affine IAM)**.** *For any almost purely affine term $v : o$, the output of* APAIAM(v) *is the unique $\tau \in \mathrm{Tree}(\Sigma)$ such that $\widetilde{\tau}$ is the normal form of v.*

The simulation by tree-walking transducers then follows the same pattern than in the purely affine case (except that we do not get reversibility).

5 From the almost !-depth 1 IAM to invisible pebbles

Now that we have seen how to prove Theorem 1.4, let us apply the same methodology to the almost !-depth 1 case, with another variation on the Interaction Abstract Machine.

The key challenge is that we can no longer rule out positions of the form $C\langle \texttt{let}\ !x = \overline{u}\ \texttt{in}\ t\rangle$ for the IAM token. If x has multiple occurrences in t, we need some information to know which of these occurrences we should move to. The standard solution to this problem is to enrich the IAM configurations with another data structure – cf. the "boxes stack" of [14] or the "log" of [1]. In our simple low-depth case, a stack of variable occurrences will be enough.

The almost !-depth 1 IAM. Let $v : o$ be an almost !-depth 1 term. To compute its normal form, we introduce a tree-generating machine whose configurations are of the form $(C\langle \underline{t}\rangle, T, L)$ or $(C\langle \overline{t}\rangle, T, L)$, where $C\langle t\rangle = v$, $T \in \{\bullet, \circ, l\}^*$, and *logs* L and logged positions l are defined by mutual induction as follows (please notice that $n \in \{0, 1\}$ for !-depth 1 terms):

$$l ::= (D_n, L_n) \qquad L_0 ::= \varepsilon \qquad L_n ::= l \cdot L_{n-1}$$

The initial configuration is $(\underline{v}, \varepsilon, \varepsilon)$. To define the computation-step function, we start by reusing all the rules of the almost purely affine IAM (Definition 4.8) except the $\underline{x}$-rule for let-bound x, and the one for !-boxes, adapting them so that they do not change the log L. We then add the rules below, *where* C_i *ranges over contexts of depth* $i \in \{0, 1\}$ (as defined in Section 2) and $A \neq o$. Please notice that transition rules now depend also on the *type* of the current subterm, indeed we have to distinguish between the "almost" and the "depth-1" exponentials:

$$\begin{aligned}
(C_0\langle \texttt{let}\ !x = u\ \texttt{in}\ D_n\langle \underline{x^{!A}}\rangle\rangle, T, L_n) &\mapsto (C_0\langle \texttt{let}\ !x = \underline{u^{!A}}\ \texttt{in}\ D_n\langle x\rangle\rangle, (D_n, L_n)\cdot T, \varepsilon)\\
(C_0\langle \texttt{let}\ !x = \overline{u^{!A}}\ \texttt{in}\ D_n\langle x\rangle\rangle, (D_n, L_n)\cdot T, \varepsilon) &\mapsto (C_0\langle \texttt{let}\ !x = u\ \texttt{in}\ D_n\langle \overline{x^{!A}}\rangle\rangle, T, L_n)\\
(C\langle \texttt{let}\ !x = u\ \texttt{in}\ D_n\langle \underline{x^{!o}}\rangle\rangle, T, L_n\cdot L) &\mapsto (C\langle \texttt{let}\ !x = \underline{u^{!o}}\ \texttt{in}\ D_n\langle x\rangle\rangle, T, L) \quad \text{(non-reversible)}\\
(C_0\langle !\underline{t^{!A}}\rangle, l\cdot T, \varepsilon) &\mapsto (C_0\langle !\underline{t^{A}}\rangle, T, l) \qquad (C_0\langle !\overline{t^{A}}\rangle, T, l) \mapsto (C_0\langle \overline{!t^{!A}}\rangle, l\cdot T, \varepsilon)\\
(C\langle \underline{!t^{!o}}\rangle, T, L) &\mapsto (C\langle !\underline{t^{o}}\rangle, T, L) \quad \text{(non-reversible)}
\end{aligned}$$

Again, this device, which is a just a specialization of the standard IAM (but again non-reversible in order to handle linearly the almost affine terms), successfully normalizes almost !-depth 1 terms of base type. Next, we would like to simulate it by some automaton model, to get a counterpart of Theorem 1.4 in this setting. The problem now is that both the log L and the tape T do not fit into the finite state of a tree-walking transducer, since their size cannot be statically bounded. Therefore, we need to target a more powerful machine model.

Invisible pebbles. Luckily, a suitable device has already been introduced by Engelfriet, Hoogeboom and Samwel [18]: the *invisible pebble tree transducer* (IPTT). Informally, it is a TWT extended with the ability to put down pebbles on input nodes. The pebbles have colors that are taken in a finite set. They can be later examined and removed: an IPTT can check whether the *last pebble to have been put down* is on the current position, and if so, it can observe its color, and perhaps decide to remove it. The "invisible" part means that only the last pebble can be seen. Thus, the lifetimes of the pebbles follow a *stack discipline* (last put down, first removed). The number of pebbles used in a computation may be unbounded.

▶ **Definition 5.1** ([18]). *An* invisible pebble tree transducer $\mathrm{Tree}(\Gamma) \rightharpoonup \mathrm{Tree}(\Sigma)$ *is made of:*

- *a finite set of* states Q *with an* initial state $q_0 \in Q$
- *a finite set of* colors $\mathfrak{C}$
- *a (partial)* transition function *that sends tuples consisting of*
  - *an input letter* $a \in \Gamma$*, a state* $q \in Q$*, a provenance* $p \in \{\downarrow_\bullet, \circlearrowleft, \uparrow_1^\bullet, \dots, \uparrow_{\mathrm{rk}(a)}^\bullet\}$
  - *a boolean* `isRoot` *which must be false if* $p = \downarrow_\bullet$
  - *a value* z *which is either a color in* $\mathfrak{C}$ *or the symbol* None

  to $\mathrm{Tree}(\Sigma,\ Q \times (\{\underbrace{\uparrow_\bullet}_{\text{prohibited if isRoot is true}}, \circlearrowleft, \downarrow_1^\bullet, \dots, \downarrow_k^\bullet, \underbrace{\mathsf{remove}}_{\text{prohibited if } z = \mathsf{None}}\} \cup \{\mathsf{put}_\mathfrak{c} \mid \mathfrak{c} \in \mathfrak{C}\}))$

Note that removing z in the arguments and $\mathsf{remove}/\mathsf{put}_\mathfrak{c}$ in the codomain would just yield an alternative presentation of tree-walking transducers (Definition 3.5).

The set of configurations of an invisible pebble tree transducer on an input tree τ is

$$\underbrace{Q \times \{\downarrow_\bullet, \circlearrowleft, \uparrow_1^\bullet, \dots\} \times \{\text{nodes of } \tau\}}_{\text{TWT configuration}} \times \underbrace{(\mathfrak{C} \times \{\text{nodes of } \tau\})^*}_{\text{pebble stack}}$$

The transition function of the IPTT determines a computation-step function by extending Definition 3.5 in the expected way (the transducer stays at the same node after a remove or $\mathsf{put}_\mathfrak{c}$ instruction). Here is an example of a configuration over $\tau = a_1(b_2(c_3), c_4)$ for some invisible pebble tree transducer: $(q, \downarrow_\bullet, c_3, [(\mathfrak{a}, b_2), (\mathfrak{b}, c_3)])$. The top of the stack is the leftmost element the list: it is an $\mathfrak{a}$-colored pebble on position b_2. Since this differs from the current node c_3, the transducer does not see any pebble ($z = \mathsf{None}$) even though there is a $\mathfrak{b}$-colored pebble on c_3 further down the stack. If we execute the instruction $\mathsf{put}_\mathfrak{a}$ while transitioning to state q', we get the configuration $(q', \circlearrowleft, c_3, [(\mathfrak{a}, c_3), (\mathfrak{a}, b_2), (\mathfrak{b}, c_3)])$. In that new configuration, the IPTT now sees the topmost pebble ($z = \mathfrak{a}$) and is thus allowed to remove it.

Outcome of the simulation. We use a single stack implementation (detailed in the technical report [38]) of the IAM presented above to compile λ-transducers to IPTTs as in the previous section, thus establishing the following comparison in expressive power:

▶ **Lemma 5.2.** *Almost !-depth 1* λ*-transducer* $\subseteq$ *invisible pebble tree transducer* $\equiv$ *MSOT-S²*.

Here, the equivalence between IPTT and MSOT-S² is a rephrasing of a result of Engelfriet et al. [18, Theorem 53], as explained in [35, §3.3].

6 Expressiveness of λ-transducers with preprocessing

Now, let us prove Theorems 1.5 and 1.7. We first note that the left-to-right inclusions are immediate consequences of Theorem 1.4 and Lemma 5.2 combined with the following facts:

- MSOT-S (and, therefore, MSOT-S²) are closed under precomposition by MSO relabeling (cf. [7, Section 3] where MSOT-S are called "MSO term graph transductions");
- TWT $\subset$ MSOT-S (a slight variant of [7, Theorem 9], cf. [35, §3.2]);
- TWT of linear growth $\subset$ MSOT (see e.g. [19, §6.2]), and purely affine λ-transducers have linear growth because the size of purely affine terms is non-increasing during β-reduction.

("f has linear growth" means that $|f(t)| = O(|t|)$.) Next, we turn to the converse inclusions.

MSOT $\subseteq$ purely affine λ-transducer $\circ$ MSO relabeling. We derive this from the results of Gallot, Lemay and Salvati [22, 23]. First, we introduce a slight generalization of their machine model for MSOTs [22, §2.3]. Their model involves bottom-up regular lookahead, but as usual in automata theory, this feature can be simulated by preprocessing by an MSO relabeling; this is why we do not include it in our version.

▶ **Definition 6.1.** *A* GLS-transducer $\mathrm{Tree}(\Gamma) \to \mathrm{Tree}(\Sigma)$ *consists of:*

- *a finite set Q of states, with a family $(A_q)_{q\in Q}$ of purely affine types;*
- *an initial state $q_0 \in Q$ and an output term $u : A_{q_0} \multimap o$ – which, like all the terms t below, may use constants $c : o^{\mathrm{rk}(c)} \multimap o$ for $c \in \Sigma$;*
- *for each $q \in Q$ and $a \in \Gamma$, a rule $q\langle a(x_1, \dots, x_{\mathrm{rk}(a)})\rangle \to t\ q_1\langle x_1\rangle\ \dots\ q_{\mathrm{rk}(a)}\langle x_{\mathrm{rk}(a)}\rangle$ where the q_i and $t : A_{q_1} \multimap \dots \multimap A_{q_{\mathrm{rk}(a)}} \multimap A_q$ are chosen depending on (q, a).*

The semantics is that a GLS-transducer performs a top-down traversal $q_0\langle\tau\rangle \to^* \tau^{\Downarrow}$ of its input tree τ which builds a λ-term $\tau^{\Downarrow} : A_{q_0}$. The normal form of $u\ \tau^{\Downarrow}$ then encodes the output tree. Our model is a bit more syntactically permissive than that of Gallot et al. (theirs would correspond to using linear rather than affine terms, and forcing u to be $\lambda x.\, x$ – so $A_{q_0} = o$); therefore, it can compute at least everything that their model can:

▶ **Theorem 6.2** (from [22, Theorem 3]). *MSOT $\subseteq$ GLS-transducer $\circ$ MSO relabeling.*

We derive our desired result on λ-transducers in two steps.

- Every GLS-transducer can be made "type-constant": $\exists A : \forall q,\ A_q = A$. This uses an encoding trick, detailed in the technical report [38], that preserves pure affineness, but not linearity.
- A type-constant GLS-transducer can be turned into an MSO relabeling (that adds to each node its top-down propagated state) followed by a purely affine λ-transducer (which is just a GLS-transducer with $|Q| = 1$).

From Theorem 6.2, we thus get MSOT $\subseteq$ purely affine λ-transducer $\circ$ (MSO relabeling)^2, and since MSO relabelings are closed under composition [7, §3], we are done.

MSOT-S $\subseteq$ almost purely affine λ-transducer $\circ$ MSO relabeling. Following the same recipe as above, we reduce this to Gallot et al.'s characterization of MSOT-S [22, Theorem 3]. Since they use Kanazawa's almost linear λ-terms [28] (which we discussed in the introduction), we need to translate such terms into our almost purely affine terms. Let us introduce the abbreviation $\lambda!x.\, t = (\lambda y.\, \mathtt{let}\ !x = y\ \mathtt{in}\ t)$ where y is a fresh variable. We define inductively: $?c = \lambda!x_1.\ \dots\ \lambda!x_{\mathrm{rk}(c)}.\, !(c\, x_1\ \dots\ x_{\mathrm{rk}(c)})$ for constants c in a ranked alphabet Σ, and

$$?x = \begin{cases} !x & \text{if } x : o \\ x & \text{otherwise} \end{cases} \qquad ?(\lambda x.\, t) = \begin{cases} \lambda!x.\, ?t & \text{if } x : o \\ \lambda x.\, ?x & \text{otherwise} \end{cases} \qquad ?(t\, u) = (?t)\, (?u)$$

▷ Claim 6.3. Let $t : A$ be almost affine as defined in [27]. Then $?t$ is almost purely affine, with type $A\{o := !o\}$. When $t : o$, if $t \longrightarrow_\beta^* \widetilde{\tau}$ (for some $\tau \in \mathrm{Tree}(\Sigma)$) then $?t \longrightarrow_\beta^* !\widetilde{\tau}$.

The inductive rule for application implies that $?(u\ \widehat{\tau}((t_a)_{a\in\Gamma})) = (?u)\ \widehat{\tau}((?t_a)_{a\in\Gamma})$, so the above claim allows us to translate λ-transducers using almost affine terms *à la* Kanazawa.

MSOT-S² ⊆ almost !-depth 1 λ-transducer ∘ relabeling. Having just finished proving Theorem 1.5 above, we may use it right away:

$$\begin{aligned}\text{MSOT-S}^2 &\equiv \text{almost purely affine } \lambda\text{-transducer} \circ \text{MSO relabeling} \circ \text{MSOT-S} && \text{(Thm. 1.5)}\\ &\equiv \text{almost purely affine } \lambda\text{-transducer} \circ \text{MSOT-S} && (*)\\ &\equiv (\text{almost purely affine } \lambda\text{-transducer})^2 \circ \text{MSO relabeling} && \text{(Thm. 1.5)}\end{aligned}$$

The line $(*)$ above relies on the fact that MSOT-S are closed under *post*composition by MSO relabelings, cf. [35, §3.3]. To conclude, we apply the composition property of λ-transducers (Proposition 1.8), noting that if A and B are almost purely affine types, then $A\{o := B\}$ is almost !-depth 1 (indeed, every '!' in it is applied to either o or B).

7 Conclusion

In this paper, we established several expressivity results relating a typed λ-calculus to tree transducers. This can be seen as furthering Nguyễn and Pradic's "implicit automata" research programme [37], even though the formal setting is slightly different; indeed, we settle one of their conjectures in Corollary 1.2. From a purely automata-theoretic perspective, our characterization of MSOT-S² is the first that involves a "one-way" device, performing a single bottom-up pass on its input (modulo preprocessing).

The equivalences between "one-way" λ-transducers and tree-walking / invisible pebble tree transducers can be seen as a trade-off between a sophisticated memory (higher-order data) and freedom of movement on the input (tree-walking reading head). This is arguably a sort of qualitative space/time trade-off (more movement means more computation steps). This is similar to the reasons that led the Geometry of Interaction to be used in implicit computational complexity when dealing with space complexity classes – an application area pioneered by Schöpp [44, 45] and leading to several further works [13, 31]. These successes even led to the belief that the GoI should give a reasonable space cost model, that is to say comparable with the one of Turing machines; but this belief is now known to be wrong in the general case of the untyped λ-calculus [2].

7.1 More related work

Katsumata [29] has connected a categorical version of the GoI (the "Int-construction") to attribute grammars [30], which are "essentially [a] notational variation" on tree-walking transducers (quoting Courcelle & Engelfriet [12, §8.7]). Recently, Pradic and Price [40] have used a "planar" version of this categorical GoI in order to prove an "implicit automata" theorem. Further GoI-automata connections of this kind are discussed in [36, §1.1].

Our methodology of connecting λ-calculus and automata via abstract machines may be compared to Salvati and Walukiewicz's [43] use of Krivine machines in the theory of higher-order recursion schemes. Clairambault and Murawski [11] also compile affine recursion schemes to automata using a game semantics that can be seen as a denotational counterpart of the operational Interaction Abstract Machine. Ghica exploited ideas from the GoI and game semantics, to design a compiler from a higher-order functional language directly to digital circuits [24], in particular targeting Mealy machines.

▶ Remark 7.1. The aforementioned work [11] is a rare example of application of some GoI variant to a setting that features the *additive connectives* of linear logic. It yields a translation from λ-terms to infinite-state systems, making it unsuited to our purposes. In most versions of the GoI, the support for additives is not as satisfactory as their handling of additive-free linear logic – a well-known issue in the linear logic community.

This obstruction also motivated our choice, discussed in the introduction, to follow the approach of Gallot et al. [22, 23] rather than Nguyễn and Pradic's "implicit automata" [37, 34]. Indeed, the latter's solution to overcome the limitations evidenced by Corollary 1.2 is to work with a linear λ-calculus with additive connectives, enabling more flexible linear usage patterns. This is analogous (see [34, Remark 6.0.1] for an actual technical connection) to moving from "strongly single use" to "single use" macro tree transducers [20, Section 5].

Salvati has shown [41] that the string languages defined by abstract categorial grammars, which are very close to our purely linear λ-transducers, coincide with the output languages of tree-to-string tree-walking transducers. He explains in his habilitation thesis [42, §3.2] that the proof ideas are similar to a game semantics of multiplicative linear logic – and the latter is closely related to GoI, as mentioned above. It would be interesting to understand to which extent his approach implicitly resembles ours, despite a very different presentation.

We also note that in the same paper that introduces almost linear λ-terms [28], Kanazawa studies a notion of "links in typed λ-terms" that looks like a form of GoI. However, these links are only well-behaved for λ-terms in normal form, while the Interaction Abstract Machine does not have this drawback. Finally, let us stress that our use of the IAM has the advantage, compared to the aforementioned works [29, 11, 41, 26], of adapting to the presence of the exponential modality "!". This is crucial in our proof of Theorem 1.7.

7.2 Perspectives

The obvious direction for further work is to study the MSOT-S^{k+1} and almost !-depth k hierarchies for $k \geq 2$. While the argument at the end of Section 6 easily generalizes to show that the former is included in the latter, we have no reason to believe that they coincide. As a more modest conjecture ("!-depth 1" means "no nested '!'s"):

▶ **Conjecture 7.2.** *!-depth 1 λ-transducer $\circ$ MSO relabeling $\equiv$ MSOT $\circ$ MSOT-S.*

We believe that the *reversible* tree-walking transducers that we have introduced also deserve to be studied further. Indeed, we expect that they should be closed under composition (cf. [15] over strings) and verify the "single-use restriction" of [12, §8.2]; the latter would imply that they can be translated into MSO transductions.

References

1 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The machinery of interaction. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020*, pages 4:1–4:15. ACM, 2020. doi:10.1145/3414080.3414108.

2 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The space of interaction. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021. doi:10.1109/LICS52264.2021.9470726.

3 Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 381–395. Springer, 2010. doi:10.1007/978-3-642-15205-4_30.

4 Rajeev Alur and Loris D'Antoni. Streaming Tree Transducers. *Journal of the ACM*, 64(5):1–55, August 2017. doi:10.1145/3092842.

5 Toshiyasu Arai. 10th Asian Logic Conference. *The Bulletin of Symbolic Logic*, 15(2):246–265, 2009. doi:10.2178/bsl/1243948490.

6 Andrew Barber. Dual Intuitionistic Linear Logic. Technical report ECS-LFCS-96-347, LFCS, University of Edinburgh, 1996. URL: http://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/.

7 Roderick Bloem and Joost Engelfriet. A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars. *Journal of Computer and System Sciences*, 61(1):1–50, August 2000. doi:10.1006/jcss.1999.1684.

8 Mikołaj Bojańczyk. Who to cite: MSO transductions, December 2019. URL: https://web.archive.org/web/20230810161232/https://www.mimuw.edu.pl/~bojan/posts/who-to-cite-mso-transductions.

9 Mikołaj Bojańczyk and Thomas Colcombet. Tree-walking automata do not recognize all regular languages. *SIAM Journal on Computing*, 38(2):658–701, 2008. doi:10.1137/050645427.

10 Mikołaj Bojańczyk and Amina Doumane. First-order tree-to-tree functions, 2020. Corrected version with erratum of a LICS 2020 paper. arXiv:2002.09307v2.

11 Pierre Clairambault and Andrzej S. Murawski. On the Expressivity of Linear Recursion Schemes. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.MFCS.2019.50.

12 Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic. A language-theoretic approach.* Encyclopedia of Mathematics and its applications, Vol. 138. Cambridge University Press, June 2012. Collection Encyclopedia of Mathematics and Applications, Vol. 138. URL: https://hal.archives-ouvertes.fr/hal-00646514.

13 Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space-bounded functional programming. *Information and Computation*, 248:150–194, 2016. doi:10.1016/j.ic.2015.04.006.

14 Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Computer Science*, 227(1):79–97, September 1999. doi:10.1016/S0304-3975(99)00049-3.

15 Luc Dartois, Paulin Fournier, Ismaël Jecker, and Nathan Lhote. On reversible transducers. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 113:1–113:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.113.

16 Joost Engelfriet. The time complexity of typechecking tree-walking tree transducers. *Acta Informatica*, 46(2):139–154, 2009. doi:10.1007/s00236-008-0087-y.

17 Joost Engelfriet. Context-free grammars with storage, 2014. Revised version of a 1986 technical report. arXiv:1408.0683.

18 Joost Engelfriet, Hendrik Jan Hoogeboom, and Bart Samwel. XML navigation and transformation by tree-walking automata and transducers with visible and invisible pebbles. *Theoretical Computer Science*, 850:40–97, January 2021. doi:10.1016/j.tcs.2020.10.030.

19 Joost Engelfriet, Kazuhiro Inaba, and Sebastian Maneth. Linear-bounded composition of tree-walking tree transducers: linear size increase and complexity. *Acta Informatica*, 58(1-2):95–152, 2021. doi:10.1007/s00236-019-00360-8.

20 Joost Engelfriet and Sebastian Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Information and Computation*, 154(1):34–91, October 1999. doi:10.1006/inco.1999.2807.

21 Joost Engelfriet and Heiko Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26(1/2):131–192, 1988. doi:10.1007/BF02915449.

22 Paul Gallot, Aurélien Lemay, and Sylvain Salvati. Linear high-order deterministic tree transducers with regular look-ahead. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 38:1–38:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.MFCS.2020.38.

23 Paul D. Gallot. *Safety of transformations of data trees: tree transducer theory applied to a verification problem on shell scripts*. PhD thesis, Université de Lille, December 2021. URL: https://theses.hal.science/tel-03773108.

24 Dan R. Ghica. Geometry of Synthesis: A Structured Approach to VLSI Design. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 363–375. ACM, 2007. doi:10.1145/1190216.1190269.

25 Tsutomu Kamimura and Giora Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981. doi:10.1016/S0019-9958(81)90438-1.

26 Makoto Kanazawa. A lambda calculus characterization of MSO definable tree transductions, September 2008. Talk given at the 10th Asian Logic Conference, Kobe University, Japan. Slides available at https://makotokanazawa.ws.hosei.ac.jp/talks/asian_logic.pdf. Abstract available at [5, p. 250–251].

27 Makoto Kanazawa. Almost affine lambda terms. In Andrzej Indrzejczak, Janusz Kaczmarek, and Michał Zawidzki, editors, *Trends in Logic XIII. Gentzen's and Jaśkowski's Heritage. 80 Years of Natural Deduction and Sequent Calculi*, pages 131–148. Wydawnictwo Uniwersytetu Łódzkiego, 2014.

28 Makoto Kanazawa. Parsing and generation as datalog query evaluation. *IfCoLog Journal of Logics and their Applications (FLAP)*, 4(4), 2017. Long version of an ACL 2007 paper. URL: http://www.collegepublications.co.uk/downloads/ifcolog00013.pdf.

29 Shin-ya Katsumata. Attribute grammars and categorical semantics. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II – Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2008. doi:10.1007/978-3-540-70583-3_23.

30 Donald E. Knuth. The genesis of attribute grammars. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications, International Conference WAGA, Paris, France, September 19-21, 1990, Proceedings*, volume 461 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1990. doi:10.1007/3-540-53101-7_1.

31 Damiano Mazza. Simple parsimonious types and logarithmic space. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPIcs*, pages 24–40. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.CSL.2015.24.

32 Damiano Mazza. *Polyadic Approximations in Logic and Computation*. Habilitation à diriger des recherches, Université Paris XIII (Sorbonne Paris Nord), November 2017. URL: https://theses.hal.science/tel-04238579.

33 Anca Muscholl and Gabriele Puppis. The Many Facets of String Transducers. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.STACS.2019.2.

34 Lê Thành Dũng Nguyễn. *Implicit automata in linear logic and categorical transducer theory*. PhD thesis, Université Paris XIII (Sorbonne Paris Nord), December 2021. URL: https://theses.hal.science/tel-04132636.

35 Lê Thành Dũng Nguyễn. Two or three things i know about tree transducers, 2024. arXiv:2409.03169.

36 Lê Thành Dũng Nguyễn, Camille Noûs, and Cécilia Pradic. Two-way automata and transducers with planar behaviours are aperiodic, 2023. arXiv:2307.11057.

37 Lê Thành Dũng Nguyễn and Cécilia Pradic. Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 135:1–135:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.135.

38 Lê Thành Dũng Nguyễn and Gabriele Vanoni. Slightly non-linear higher-order tree transducers. *CoRR*, abs/2402.05854, 2024. doi:10.48550/arXiv.2402.05854.

39 Alexander Okhotin. Graph-walking automata: From whence they come, and whither they are bound. In Michal Hospodár and Galina Jirásková, editors, *Implementation and Application of Automata – 24th International Conference, CIAA 2019, Košice, Slovakia, July 22-25, 2019, Proceedings*, volume 11601 of *Lecture Notes in Computer Science*, pages 10–29. Springer, 2019. doi:10.1007/978-3-030-23679-3_2.

40 Cécilia Pradic and Ian Price. Implicit automata in λ-calculi iii: affine planar string-to-string functions. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 4 – Proceedings of MFPS XL, December 2024. doi:10.46298/entics.14804.

41 Sylvain Salvati. Encoding second order string ACG with deterministic tree walking transducers. In Shuly Wintner, editor, *The 11th conference on Formal Grammar*, FG Online Proceedings, pages 143–156, Malaga, Spain, 2006. Paola Monachesi; Gerald Penn; Giorgio Satta; Shuly Wintner, CSLI Publications. URL: https://web.stanford.edu/group/cslipublications/cslipublications/FG/2006/salvati.pdf.

42 Sylvain Salvati. *Lambda-calculus and formal language theory*. Habilitation à diriger des recherches, Université de Bordeaux, December 2015. URL: https://theses.hal.science/tel-01253426.

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

44 Ulrich Schöpp. Space-efficient computation by interaction. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 606–621. Springer, 2006. doi:10.1007/11874683_40.

45 Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 411–420. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.45.

46 Gabriele Vanoni. *On Reasonable Space and Time Cost Models for the λ-Calculus*. PhD thesis, Alma Mater Studiorum – Università di Bologna, June 2022. doi:10.48676/unibo/amsdottorato/10276.