On Computability of Data Word Functions Defined by Transducers

Léo Exibard, Emmanuel Filiot, Pierre-Alain Reynier

On Computability of Data Word Functions Defined by Transducers

Léo Exibard*[1,2], Emmanuel Filiot†[1], and Pierre-Alain Reynier‡[2]

[1]Université Libre de Bruxelles, Brussels, Belgium
[2]Aix-Marseille Université, Marseille, France

Abstract

In this paper, we investigate the problem of synthesizing computable functions of infinite words over an infinite alphabet (data ω-words). The notion of computability is defined through Turing machines with infinite inputs which can produce the corresponding infinite outputs in the limit. We use non-deterministic transducers equipped with registers, an extension of register automata with outputs, to specify functions. Such transducers may not define functions but more generally relations of data ω-words, and we show that it is PSPACE-complete to test whether a given transducer defines a function. Then, given a function defined by some register transducer, we show that it is decidable (and again, PSPACE-c) whether such function is computable. As for the known finite alphabet case, we show that computability and continuity coincide for functions defined by register transducers, and show how to decide continuity. We also define a subclass for which those problems are PTIME.

Keywords: Data Words · Register Automata · Register Transducers · Functionality · Continuity · Computability.

1 Introduction

Context Program synthesis aims at deriving, in an automatic way, a program that fulfils a given specification. Such setting is very appealing when for instance the specification describes, in some abstract formalism (an automaton or ideally a logic), important properties that the program must satisfy. The synthesised program is then *correct-by-construction* with regards to those properties. It is

*Corresponding author: leo.exibard@ulb.ac.be. Funded by a FRIA fellowship from the F.R.S.-FNRS.

†Research associate of F.R.S.-FNRS. He is supported by the ARC Project Transform Fédération Wallonie-Bruxelles and the FNRS CDR J013116F and MIS F451019F projects.

‡Partly funded by the DeLTA project (ANR-16-CE40-0007).

particularly important and desirable for the design of safety-critical systems with hard dependability constraints, which are notoriously hard to design correctly.

Program synthesis is hard to realise for general-purpose programming languages but important progress has been made recently in the automatic synthesis of *reactive systems*. In this context, the system continuously receives input signals to which it must react by producing output signals. Such systems are not assumed to terminate and their executions are usually modelled as infinite words over the alphabets of input and output signals. A specification is thus a set of pairs $(\mathsf{in},\mathsf{out})$, where in and out are infinite words, such that out is a legitimate output for in. Most methods for reactive system synthesis only work for *synchronous* systems over *finite* sets of input and output signals Σ and Γ. In this synchronous setting, input and output signals alternate, and thus *implementations* of such a specification are defined by means of *synchronous* transducers, which are Büchi automata with transitions of the form (q, σ, γ, q'), expressing that in state q, when getting input $\sigma \in \Sigma$, output $\gamma \in \Gamma$ is produced and the machine moves to state q'. We aim at building *deterministic* implementations, in the sense that the output γ and state q' uniquely depend on q and σ. The realisability problem of specifications given as synchronous non-deterministic transducers, by implementations defined by synchronous deterministic transducers is known to be decidable [14, 20]. In this paper, we are interested in the *asynchronous* setting, in which transducers can produce none or several outputs at once every time some input is read, i.e., transitions are of the form (q, σ, w, q') where $w \in \Gamma^*$. However, such generalisation makes the realisability problem undecidable [2, 9].

Synthesis of Transducers with Registers In the above setting, the set of signals is considered to be finite. This assumption is not realistic in general, as signals may come with unbounded information (e.g. process ids) that we call here *data*. To address this limitation, recent works have considered the synthesis of reactive systems processing *data words* [17, 6, 16, 7]. Data words are infinite words over an alphabet $\Sigma \times \mathcal{D}$, where Σ is a finite set and $\mathcal{D}$ is a possibly infinite countable set. To handle data words, just as automata have been extended to *register automata*, transducers have been extended to *register transducers*. Such transducers are equipped with a finite set of registers in which they can store data and with which they can compare data for equality or inequality. While the realisability problem of specifications given as synchronous non-deterministic register transducers ($\mathsf{NRT}_{\mathsf{syn}}$) by implementation defined by synchronous deterministic register transducers ($\mathsf{DRT}_{\mathsf{syn}}$) is undecidable, decidability is recovered for specifications defined by universal register transducers and by giving as input the number of registers the implementation must have [7, 17].

Computable Implementations In the previously mentioned works, both for finite or infinite alphabets, implementations are considered to be deterministic transducers. Such an implementation is guaranteed to use only a constant amount of memory (assuming data have size $O(1)$). While it makes sense with regards to memory-efficiency, some problems turn out to be undecidable, as already mentioned: realisability of $\mathsf{NRT}_{\mathsf{syn}}$ specifications by $\mathsf{DRT}_{\mathsf{syn}}$, or, in the finite alphabet setting, when both the specification and implementation are asynchronous. In this paper, we propose to study computable implementations, in the sense of (partial) functions f of data ω-words computable by some Turing machine M that has an infinite input $x \in \mathrm{dom}(f)$, and must produce longer and longer prefixes of the output $f(x)$ as it reads longer and longer prefixes of the input x. Therefore, such a machine produces the output $f(x)$ in the limit. We denote by TM (for Turing machine) the class of Turing machines computing functions in this sense. As an example, consider the function f that takes as input any data ω-word $u = (\sigma_1, d_1)(\sigma_2, d_2)\dots$ and outputs $(\sigma_1, d_1)^\omega$ if d_1 occurs at least twice in u, and otherwise outputs u. This function is not computable, as an hypothetic machine could not output anything as long as d_1 is not met a second time. However, the following function g is computable. It is defined only on words $(\sigma_1, d_1)(\sigma_2, d_2)\dots$ such that $\sigma_1\sigma_2 \cdots \in ((a+b)c^*)^\omega$, and transforms any (σ_i, d_i) by (σ_i, d_1) if the next symbol in $\{a, b\}$ is an a, otherwise it keeps (σ_i, d_i) unchanged. To compute it, a TM would need to store d_1, and then wait until the next symbol in $\{a, b\}$ is met before outputting something. Since the finite input labels are necessarily in $((a+b)c^*)^\omega$, this machine will produce the whole output in the limit. Note that g cannot be defined by any deterministic register transducer, as it needs unbounded memory to be implemented.

However, already in the finite alphabet setting, the problem of deciding if a specification given as some non-deterministic synchronous transducer is realisable by some computable function is open. The particular case of realisability by computable functions of universal domain (the set of all ω-words) is known to be decidable [12]. In the asynchronous setting, the undecidability proof of [2] can be easily adapted to show the undecidability of realisability of specifications given by non-deterministic (asynchronous) transducers by computable functions.

Functional Specifications As said before, a specification is in general a relation from inputs to outputs. If this relation is a function, we call it functional. Due to the negative results just mentioned about the synthesis of computable functions from non-functional specifications, we instead here focus on the case of functional specifications and address the following general question: given the specification of a function of data ω-words, is this function "implementable", where we define "implementable" as "being computable by some Turing machine". Moreover, if it is implementable, then we want a procedure to automatically generate an algorithm that computes it. This raises another important question: how to decide whether a specification is functional ? We investigate these questions for asynchronous register transducers, here called register transducers. This asynchrony allows for much more expressive power, but is a source of technical challenge.

Contributions In this paper, we solve the questions mentioned before for the class of (asynchronous) non-deterministic register transducers (NRT). We also give fundamental results on this class. In particular, we prove that:

1. deciding whether an NRT defines a function is PSpace-complete,
2. deciding whether two functions defined by NRT are equal on the intersection of their domains is PSpace-complete,
3. the class of functions defined by NRT is effectively closed under composition,
4. computability and continuity are equivalent notions for functions defined by NRT, where continuity is defined using the classical Cantor distance,
5. deciding whether a function given as an NRT is computable is PSpace-c,
6. those problems are in PTime for a subclass of NRT, called test-free NRT.

Finally, we also mention that considering the class of deterministic register transducers (DRT for short) instead of computable functions as a yardstick for the notion of being "implementable" for a function would yield undecidability. Indeed, given a function defined by some NRT, it is in general undecidable to check whether this function is realisable by some DRT, by a simple reduction from the universality problem of non-deterministic register automata [19].

Related Work The notion of continuity with regards to Cantor distance is not new, and for rational functions over finite alphabets, it was already known to be decidable [21]. Its connection with computability for functions of ω-words over a finite alphabet has recently been investigated in [3] for one-way and two-way transducers. Our results lift some of theirs to the setting of data words. The model of test-free NRT can be seen as a one-way non-deterministic version of a model of two-way transducers considered in [5].

Outline In Section 2, we define register transducers and automata. In Section 3, we study functions defined by NRT: we show that their functionality problem is decidable and that they are closed under composition. In Section 4, we connect computability and continuity for those functions, and prove that these notions are decidable. Finally, we study the test-free restriction in Section 5.

2 Data Words and Register Transducers

For a (possibly infinite) set S, we denote by S^* (resp. S^ω) the set of finite (resp. infinite) words over this alphabet, and we let $S^\infty = S^* \cup S^\omega$. For a word $u = u_1 \dots u_n$, we denote $\|u\| = n$ its length, and, by convention, for $u \in S^\omega, \|u\| = \infty$. The empty word is denoted ε. For $1 \leq i \leq j \leq \|u\|$, we let $u[i{:}j] = u_i u_{i+1} \dots u_j$ and $u[i] = u[i{:}i]$ the ith letter of u. For $u, v \in S^\infty$, we say that u is a prefix of v, written $u \preceq v$, if there exists $w \in S^\infty$ such that $v = uw$.

In this case, we define $u^{-1}v = w$. For $u, v \in S^\infty$, we say that u and v *mismatch*, written $\mathsf{mismatch}(u, v)$, when there exists a position i such that $1 \leq i \leq \|u\|$, $1 \leq i \leq \|v\|$ and $u[i] \neq v[i]$. Finally, for $u, v \in S^\infty$, we denote by $u \wedge v$ their longest common prefix, i.e. the longest word $w \in S^\infty$ such that $w \preceq u$ and $w \preceq v$.

Data Words In the whole paper, Σ and Γ are two finite alphabets and $\mathcal{D}$ is a countably infinite set of elements called *data*. We will use letter σ (resp. γ, d) to denote elements of Σ (resp. Γ, $\mathcal{D}$). We also distinguish an (arbitrary) data value $\mathsf{d}_0 \in \mathcal{D}$. Given a set R, let τ_0^R be the constant function defined by $\tau_0^R(r) = \mathsf{d}_0$ for all $r \in R$. Given a finite alphabet A, a *labelled data* is a pair $x = (a, d) \in A \times \mathcal{D}$, where a is the *label* and d the *data*. We define the projections $\mathsf{lab}(x) = a$ and $\mathsf{dt}(x) = d$. A *data word* over A and $\mathcal{D}$ is an infinite sequence of labelled data, i.e. a word $w \in (A \times \mathcal{D})^\omega$. We extend the projections lab and dt to data words naturally, i.e. $\mathsf{lab}(w) \in A^\omega$ and $\mathsf{dt}(w) \in \mathcal{D}^\omega$. A *data word language* is a subset $L \subseteq (A \times \mathcal{D})^\omega$. Note that in this paper, data words are infinite, otherwise they are called *finite data words*.

2.1 Register Transducers

Register transducers are transducers recognising data word relations. They are an extension of finite transducers to data word relations, in the same way register automata [15] are an extension of finite automata to data word languages. Here, we define them over infinite data words with a Büchi acceptance condition, and allow multiple registers to contain the same data, with a syntax close to [18]. The current data can be compared for equality with the register contents via tests, which are symbolic and defined via Boolean formulas of the following form. Given R a set of registers, a *test* is a formula ϕ satisfying the following syntax:

$$\phi \ ::= \ \top \mid \bot \mid r^{=} \mid r^{\neq} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi$$

where $r \in R$. Given a valuation $\tau : R \to \mathcal{D}$, a test ϕ and a data d, we denote by $\tau, d \models \phi$ the satisfiability of ϕ by d in valuation τ, defined as $\tau, d \models r^{=}$ if $\tau(r) = d$ and $\tau, d \models r^{\neq}$ if $\tau(r) \neq d$. The Boolean combinators behave as usual. We denote by Tst_R the set of (symbolic) tests over R.

Definition 1. A non-deterministic register transducer (NRT) is a tuple $T = (Q, R, i_0, F, \Delta)$, where Q is a finite set of *states*, $i_0 \in Q$ is the *initial* state, $F \subseteq Q$ is the set of *accepting* states, R is a finite set of *registers* and $\Delta \subseteq Q \times \Sigma \times \mathsf{Tst}_R \times 2^R \times (\Gamma \times R)^* \times Q$ is a finite set of *transitions*. We write $q \xrightarrow[T]{\sigma, \phi \mid \mathsf{asgn}, o} q'$ for $(q, \sigma, \phi, \mathsf{asgn}, o, q') \in \Delta$ (we omit T when clear from the context).

The semantics of a register transducer is given by a labelled transition system: we define $L_T = (C, \Lambda, \to)$, where $C = Q \times (R \to \mathcal{D})$ is the set of configurations, $\Lambda = (\Sigma \times \mathcal{D}) \times (\Gamma \times \mathcal{D})^*$ is the set of labels, and we have, for all $(q, \tau), (q', \tau') \in C$

and for all $(l, w) \in \Lambda$, that $(q, \tau) \xrightarrow{(l,w)} (q', \tau')$ whenever there exists a transition $q \xrightarrow[T]{\sigma,\phi|\mathsf{asgn},o} q'$ such that, by writing $l = (\sigma', d)$ and $w = (\gamma'_1, d_1) \dots (\gamma'_n, d_n)$:

- (Matching labels) $\sigma = \sigma'$
- (Compatibility) d satisfies the test $\phi \in \mathsf{Tst}_R$, i.e. $\tau, d \models \phi$.
- (Update) τ' is the successor register configuration of τ with regards to d and asgn: $\tau'(r) = d$ if $r \in \mathsf{asgn}$, and $\tau'(r) = \tau(r)$ otherwise
- (Output) By writing $o = (\gamma_1, r_1) \dots (\gamma_m, r_m)$, we have that $m = n$ and for all $1 \leq i \leq n$, $\gamma_i = \gamma'_i$ and $d_i = \tau'(r_i)$.

Then, a *run* of T is an infinite sequence of configurations and transitions $\rho = (q_0, \tau_0) \xrightarrow[L_T]{(u_1,v_1)} (q_1, \tau_1) \xrightarrow[L_T]{(u_2,v_2)} \cdots$. Its input is $\mathsf{in}(\rho) = u_1 u_2 \dots$, its output is $\mathsf{out}(\rho) = v_1 \cdot v_2 \dots$. We also define its sequence of states $\mathsf{st}(\rho) = q_0 q_1 \dots$, and its *trace* $\mathsf{tr}(\rho) = u_1 \cdot v_1 \cdot u_2 \cdot v_2 \dots$. Such run is *initial* if $(q_0, \tau_0) = (i_0, \tau_0^R)$. It is *final* if it satisfies the Büchi condition, i.e. $\inf(\mathsf{st}) \cap F \neq \varnothing$, where $\inf(\mathsf{st}) = \{q \in Q \mid q = q_i \text{ for infinitely many } i\}$. Finally, it is *accepting* if it is both initial and final. We then write $(q_0, \tau_0) \xrightarrow[T]{u|v}$ to express that there is a final run ρ of T starting from (q_0, τ_0) such that $\mathsf{in}(\rho) = u$ and $\mathsf{out}(\rho) = v$. In the whole paper, and unless stated otherwise, we always assume that the output of an accepting run is infinite ($v \in (\Gamma \times \mathcal{D})^\omega$), which can be ensured by a Büchi condition.

A *partial run* is a finite prefix of a run. The notions of input, output and states are extended by taking the corresponding prefixes. We then write $(q_0, \tau_0) \xrightarrow[T]{u|v} (q_n, \tau_n)$ to express that there is a partial run ρ of T starting from configuration (q_0, τ_0) and ending in configuration (q_n, τ_n) such that $\mathsf{in}(\rho) = u$ and $\mathsf{out}(\rho) = v$.

Finally, the relation represented by a transducer T is:

$$\llbracket T \rrbracket = \big\{(u, v) \in (\Sigma \times \mathcal{D})^\omega \times (\Gamma \times \mathcal{D})^\omega \mid \text{there exists an accepting run } \rho \text{ of } T \\ \text{such that } \mathsf{in}(\rho) = u \text{ and } \mathsf{out}(\rho) = v\big\}$$

Example 1. As an example, consider the register transducer T_{rename} depicted in Figure 1. It realises the following transformation: consider a setting in which we deal with logs of communications between a set of clients. Such a log is an infinite sequence of pairs consisting of a tag, chosen in some finite alphabet Σ, and the identifier of the client delivering this tag, chosen in some infinite set of data values. The transformation should modify the log as follows: for a given client that needs to be modified, each of its messages should now be associated with some new identifier. The transformation has to verify that this new identifier is indeed free, *i.e.* never used in the log. Before treating the log, the transformation receives as input the id of the client that needs to be modified (associated with the tag del), and then a sequence of identifiers (associated with the tag ch), ending with #. The transducer is non-deterministic as it has to guess which of these identifiers it can choose to replace the one of the client. In particular, observe that it may associate multiple output words to a same input if two such free identifiers exist.

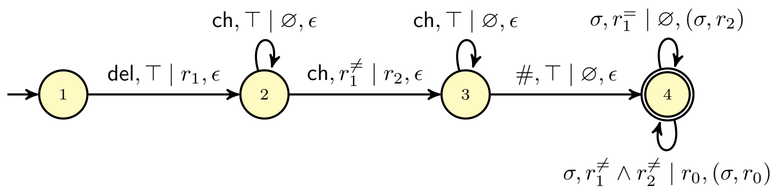

Figure 1: A register transducer T_{rename}. It has three registers r_1, r_2 and r_0 and four states. σ denotes any letter in Σ, r_1 stores the id of del and r_2 the chosen id of ch, while r_0 is used to output the last data value read as input. As we only assign data to single registers, we write r_i for the singleton assignment set $\{r_i\}$.

Finite Transducers Since we reduce the decision of continuity and functionality of NRT to the one of finite transducers, let us introduce them: a finite transducer (NFT for short) is an NRT with 0 registers (i.e. $R = \varnothing$). Thus, its transition relation can be represented as $\Delta \subseteq Q \times \Sigma \times \Gamma^* \times Q$. A direct extension of the construction of [15, Proposition 1] allows to show that:

Proposition 1. *Let T be an NRT with k registers, and let $X \subset_f \mathcal{D}$ be a finite subset of data. Then, $[\![T]\!] \cap (\Sigma \times X)^\omega \times (\Gamma \times X)^\omega$ is recognised by an NFT of exponential size, more precisely with $O(|Q| \times |X|^{|R|})$ states.*

2.2 Technical Properties of Register Automata

Although automata are simpler machines than transducers, we only use them as tools in our proofs, which is why we define them from transducers, and not the other way around. A non-deterministic register automaton, denoted NRA, is a transducer without outputs: its transition relation is $\Delta \subseteq Q \times \Sigma \times \mathsf{Tst}_R \times 2^R \times \{\varepsilon\} \times Q$ (simply represented as $\Delta \subseteq Q \times \Sigma \times \mathsf{Tst}_R \times 2^R \times Q$). The semantics are the same, except that now we lift the condition that the output v is infinite since there is no output. For A an NRA, we denote $L(A) = \{u \in (\Sigma \times \mathcal{D})^\omega \mid$ there exists an accepting run ρ of A over $u\}$. Necessarily the output of an accepting run is ε. In this section, we establish technical properties about NRA.

Theorem 2, the so-called "indistinguishability property", was shown in the seminal paper by Kaminski and Francez [15, Proposition 1]. Their model differs in that they do not allow distinct registers to contain the same data, and in the corresponding test syntax, but their result easily carries to our setting. It states that if an NRA accepts a data word, then such data word can be relabelled with data from any set containing d_0 and with at least $k + 1$ elements. Indeed, at any point of time, the automaton can only store at most k data in its registers, so its notion of "freshness" is a local one, and forgotten data can thus be reused as fresh ones. Moreover, as the automaton only tests data for equality, their actual value does not matter, except for d_0 which is initially contained in the registers.

Such "small-witness" property is fundamental to NRA, and will be paramount in establishing decidability of functionality (Section 3) and of computability (Section 4). We use it jointly with Theorem 3, which states that the interleaving of the traces of runs of an NRT can be recognised with an NRA, and Theorem 4, which expresses that an NRA can check whether interleaved words coincide on some bounded prefix, and/or mismatch before some given position.

Proposition 2 ([15]). *Let A be an* NRA *with k registers. If $L(A) \neq \varnothing$, then, for any $X \subseteq \mathcal{D}$ of size $|X| \geq k+1$ such that $\mathsf{d}_0 \in X$, $L(A) \cap (\Sigma \times X)^\omega \neq \varnothing$.*

The runs of a register transducer T can be flattened to their traces, so as to be recognised by an NRA. Those traces can then be interleaved, in order to be compared. The proofs of the following properties are quite straightforward, but are given for the sake of completeness.

Let $\rho_1 = (q_0, \tau_0) \xrightarrow[L_T]{(u_1, u_1')} (q_1, \tau_1) \dots$ and $\rho_2 = (p_0, \mu_0) \xrightarrow[L_T]{(v_1, v_1')} (p_1, \mu_1) \dots$ be two runs of a transducer T. Then, we define their *interleaving* $\rho_1 \otimes \rho_2 = u_1 \cdot u_1' \cdot v_1 \cdot v_1' \cdot u_2 \cdot u_2' \cdot v_2 \cdot v_2' \dots$ and $L_\otimes(T) = \{\rho_1 \otimes \rho_2 \mid \rho_1$ and ρ_2 are accepting runs of $T\}$.

Lemma 3. *If T has k registers, then $L_\otimes(T)$ is recognised by an* NRA *with $2k$ registers.*

Proof. We construct an automaton A with $2k$ registers which, on an input data word $u \in ((\Sigma \times \mathcal{D})(\Gamma \times \mathcal{D})^*)^\omega$ guesses, using non-determinism, a decomposition $u = u_1 \cdot u_1' \cdot v_1 \cdot v_1' \cdot u_2 \cdot u_2' \cdot v_2 \cdot v_2' \dots$ as defined before. In particular for all i, $|u_i| = |v_i| = 1$. The automaton A also maintains two copies of T (that is why A needs $2k$ registers). The first one makes sure that $u_1 u_1' u_2 u_2' \dots = \mathsf{tr}(\rho_1)$ for some run ρ_1, the other one checks that $v_1 v_1' v_2 v_2' \dots = \mathsf{tr}(\rho_2)$ for some run ρ_2. To check those two properties, A simulates T on both decompositions, using the two copies of T. For the input part (when reading u_1 or v_1), A simply makes the same tests as T does. For the output part, if A simulates a transition of T with output $o = (\sigma_1, r_1) \dots (\sigma_n, r_n)$ on some output v_i, it makes n transitions which successively check that r_j is equal to the current data, for all $1 \leq j \leq n$. ☐

Lemma 4. *Let $i, j \in \mathbb{N} \cup \{\infty\}$. We define $M_j^i = \{u_1 u_1' v_1 v_1' \dots \mid \forall k \geq 1, u_k, v_k \in (\Sigma \times \mathcal{D}), u_k', v_k' \in (\Gamma \times \mathcal{D})^*, \forall 1 \leq k \leq j, v_k = u_k$ and $\|u_1' \cdot u_2' \dots \wedge v_1' \cdot v_2' \dots\| \leq i\}$. Then, M_j^i is recognisable by an* NRA *with 2 registers and with 1 register if $i = \infty$.*

Proof. Checking that $\forall k \geq 1, u_k, v_k \in (\Sigma \times \mathcal{D}), u_k', v_k' \in (\Gamma \times \mathcal{D})^*$ is a regular property, since there are no constraints over the data. Now, checking that $\forall 1 \leq k \leq j, v_k = u_k$ requires a j-bounded counter (or no counter if $j = \infty$) which can be stored in memory, along with one register to store $\mathsf{dt}(u_k)$. Now, if $i = \infty$, there is no additional property to check. If $i < \infty$, checking that $\|u_1' \cdot u_2' \dots \wedge v_1' \cdot v_2' \dots\| \leq i\}$ requires two i-bounded counters to keep track of the respective lengths of the prefixes of $u' = u_1' \cdot u_2' \dots$ and $v' = v_1' \cdot v_2' \dots$ which have been read so far, along with one register: the automaton guesses whether the mismatch position first appears in u' or v', stores the corresponding data in its additional register, as well as the label in its memory, goes to the corresponding position in v' or u' with the help of its i-bounded counters and checks that either the data or the label indeed mismatches. □

3 Functionality, Equivalence and Composition of NRT

In general, since they are non-deterministic, NRT may not define functions but relations, as illustrated by Example 1. In this section, we first show that deciding whether a given NRT defines a function is PSPACE-complete, in which case we call it *functional*. We show, as a consequence, that testing whether two functional NRT define two functions which coincide on their common domain is PSPACE-complete. Finally, we show that functions defined by NRT are closed under composition. This is an appealing property in transducer theory, as it allows to define complex functions by composing simple ones.

Example 2. As explained before, the transducer T_{rename} described in Example 1 is not functional. To gain functionality, one can reinforce the specification by considering that one gets at the beginning a list of k possible identifiers, and that one has to select the first one which is free, for some fixed k. This transformation is realised by the register transducer T_{rename2} depicted in Figure 2 (for $k = 2$).

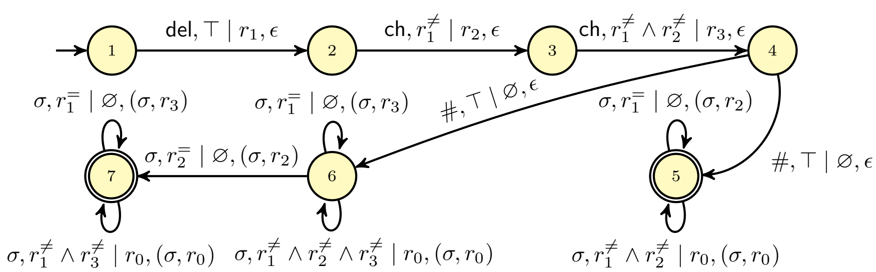

Figure 2: A NRT T_{rename2}, with four registers r_1, r_2, r_3 and r_0 (the latter being used, as in Figure 1, to output the last read data). After reading the # symbol, it guesses whether the value of register r_2 appears in the suffix of the input word. If not, it goes to state 5, and replaces occurrences of r_1 by r_2. Otherwise, it moves to state 6, waiting for an occurrence of r_2, and replaces occurrences of r_1 by r_3.

3.1 Functionality

Let us start with the functionality problem in the data-free case. It is already known that checking whether an NFT over ω-words is functional is decidable [13, 11]. By relying on the pattern logic of [10] designed for transducers of *finite* words, we can show that it is decidable in NLogSpace.

Proposition 5. *Deciding whether an* NFT *is functional is in* NLogSpace*.*

Proof. Let T be some NFT. By definition, $[\![T]\!]$ maps ω-words to ω-words, we have that T is *not* functional iff there exist three ω-words w, w_1, w_2 such that $(w, w_1) \in [\![T]\!]$, $(w, w_2) \in [\![T]\!]$ and there is a mismatch between w_1 and w_2, i.e. a position i such that $w_1[i] \neq w_2[i]$. By taking a sufficiently long prefix u of w, the latter is equivalent to the existence of finite runs $r_1 : q_1 \xrightarrow{u|v_1} p_1$ and $r_2 : q_2 \xrightarrow{u|v_2} p_2$, where q_1, q_2 are initial, p_1, p_2 are co-accessible by the same ω-words (in the sense that there exist two accepting runs from p_1 and p_2 on the same ω-word), and such that there is a mismatch between v_1 and v_2. This property is expressible in the pattern logic of [10] for transducers of finite words, whose model-checking problem is in NLogSpace. More precisely, we model-check against T the following pattern formula (using the syntax of [10]):

$$\begin{array}{l}\exists \pi_1 : q_1 \xrightarrow{u|v_1} p_1 \exists \pi_2 : q_2 \xrightarrow{u|v_2} p_2 \\ \mathsf{init}(q_1) \wedge \mathsf{init}(q_2) \wedge \mathsf{coacc}(p_1, p_2) \wedge \mathsf{mismatch}(v_1, v_2)\end{array}$$

The predicate $\mathsf{coacc}(p_1, p_2)$ is not directly defined in [10], which is a logic for transducers over finite words, but we show that it is easily definable in the pattern logic. Indeed, the property of (p_1, p_2) to be co-accessible by the same ω-word is equivalent to asking the existence of finite runs r_1', r_2' of the following form, where the outputs have not been indicated as they do not matter for this property:

$$r_1' \ : \ p_1 \xrightarrow{u_1} p_1' \xrightarrow{u_2} p_1'' \xrightarrow{u_3} p_1' \qquad r_2' \ : \ p_2 \xrightarrow{u_1} p_2' \xrightarrow{u_2} p_2'' \xrightarrow{u_3} p_2'$$

such that p_1', p_2'' are accepting states. Indeed, if (p_1, p_2) is co-accessible and since we use a Büchi accepting condition, it is co-accessible by a lasso word $u_1(u_2u_3)^\omega$ on which there is a run from p_1 with some accepting states p_1' and a run from p_2 with some accepting state p_2'', both occurring on the loop of the lasso. Therefore, $\mathsf{coacc}(p_1, p_2)$ holds true iff the following pattern logic formula is true against T (seen as a transducer over finite words):

$$\begin{array}{l}\exists \pi_1' : p_1 \xrightarrow{u_1} p_1' \exists \pi_1'' : p_1' \xrightarrow{u_2} p_1'' \exists \pi_1''' : p_1'' \xrightarrow{u_3} p_1' \\ \exists \pi_2' : p_2 \xrightarrow{u_1} p_2' \exists \pi_2'' : p_2' \xrightarrow{u_2} p_2'' \exists \pi_2''' : p_2'' \xrightarrow{u_3} p_2' \\ \mathsf{final}(p_1') \wedge \mathsf{final}(p_2'')\end{array}$$

This concludes the proof. ☐

The following theorem shows that a relation between data-words defined by an NRT with k registers is a function iff its restriction to a set of data with at most $2k + 3$ data is a function. As a consequence, functionality is decidable as it reduces to the functionality problem of transducers over a finite alphabet.

Theorem 6. *Let T be an* NRT *with k registers. Then, for all $X \subseteq \mathcal{D}$ of size $|X| \geq 2k+3$ such that $\mathsf{d}_0 \in X$, we have that T is functional if and only if $[\![T]\!] \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega)$ is functional.*

Proof. The left-to-right direction is trivial. Now, assume T is not functional. Let $x \in (\Sigma \times \mathcal{D})^\omega$ be such that there exists $y, z \in (\Gamma \times \mathcal{D})^\omega$ such that $y \neq z$ and $(x, y), (x, z) \in [\![T]\!]$. Let $i = \|y \wedge z\|$. Then, consider the language $L = \{\rho_1 \otimes \rho_2 \mid \rho_1$ and ρ_2 are accepting runs of $T, \mathsf{in}(\rho_1) = \mathsf{in}(\rho_2)$ and $\|\mathsf{out}(\rho_1) \wedge \mathsf{out}(\rho_2)\| \leq i\}$. Since, by Theorem 3, $L_\otimes(T)$ is recognised by an NRA with $2k$ registers and, by Theorem 4, M^i_∞ is recognised by an NRA with 2 registers, we get that $L = L_\otimes(T) \cap M^i_\infty$ is recognised by an NRA with $2k+2$ registers.

Now, $L \neq \varnothing$, since, by letting ρ_1 and ρ_2 be the runs of T both with input x and with respective outputs y and z, we have that $w = \rho_1 \otimes \rho_2 \in L$. Let $X \subseteq \mathcal{D}$ such that $|X| \geq 2k+3$ and $\mathsf{d}_0 \in X$. By Theorem 2, we get that $L \cap (\Sigma \times X)^\omega \neq \varnothing$. By letting $w' = \rho'_1 \otimes \rho'_2 \in L \cap (\Sigma \times X)^\omega$, and $x' = \mathsf{in}(\rho'_1) = \mathsf{in}(\rho'_2)$, $y' = \mathsf{out}(\rho'_1)$ and $z' = \mathsf{out}(\rho'_2)$, we have that $(x', y'), (x', z') \in [\![T]\!] \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega)$ and $\|y' \wedge z'\| \leq i$, so, in particular, $y' \neq z'$ (since both are infinite words). Thus, $[\![T]\!] \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega)$ is not functional. $\square$

As a consequence of Theorem 5 and Theorem 6, we obtain the following result. The lower bound is obtained by encoding non-emptiness of register automata, which is PSPACE-complete [4].

Corollary 7. *Deciding whether an* NRT *T is functional is* PSPACE*-complete.*

Proof. Let T be an NRT with k registers. Choose any $X \subseteq \mathcal{D}$ of size $2k+3$ such that $\mathsf{d}_0 \in X$. By Theorem 6, T is functional if and only if $[\![T]\!] \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega)$ is. By Theorem 1, $[\![T]\!] \cap (\Sigma \times X)^\omega \times (\Gamma \times X)^\omega$ is recognisable by an NFT with $O(|Q| \times |X|^{|R|})$ states, which can be constructed on-the-fly. The functionality problem for data-free transducers is in NLOGSPACE by Theorem 5, so this yields a PSPACE procedure.

For the hardness, it is known that the emptiness problem of a register automaton A is PSPACE-hard [4]. It can be easily reduced to a functionality problem, by constructing a transducer T which is functional iff its domain is empty, for instance a transducer realising the relation $f_1 \cup f_2$ where $f_1 = \{(wxw', wxw') \mid w \in L(A), w' \in (\Sigma \times \mathcal{D})^\omega, x \in \{\#\} \times \mathcal{D}\}$ and $f_2 = \{(wxw', wx^\omega) \mid w \in L(A), w' \in (\Sigma \times \mathcal{D})^\omega, x \in \{\#\} \times \mathcal{D}\}$, where $\# \notin \Sigma$ is a new symbol. The transducer T can be constructed from A in polynomial time, and is functional iff $L(A) = \varnothing$. $\square$

3.2 Equivalence

As a consequence of Theorem 7, the following problem on the equivalence of NRT is decidable:

Theorem 8. *The problem of deciding, given two functions f, g defined by* NRT*, whether for all $x \in \mathrm{dom}(f) \cap \mathrm{dom}(g)$, $f(x) = g(x)$, is* PSPACE*-complete.*

Proof. The formula $\forall x \in \mathrm{dom}(f) \cap \mathrm{dom}(g) \cdot f(x) = g(x)$ is true iff the relation $f \cup g = \{(x, y) \mid y = f(x) \vee y = g(x)\}$ is a function. The latter can be decided by testing whether the disjoint union of the transducers defining f and g defines a function, which is in PSpace by Theorem 7. To show the hardness, we reduce the emptiness problem of NRA A over finite words, just as in the proof of Theorem 7. In particular, the functions f_1 and f_2 defined in this proof (which have the same domain) are equal iff $L(A) = \varnothing$. □

Note that under the promise that f and g have the same domain, then the latter theorem implies that it is decidable to check whether the two functions are equal. However, checking $\mathrm{dom}(f) = \mathrm{dom}(g)$ is undecidable, as the language-equivalence problem for non-deterministic register automata is undecidable, since, in particular, universality is undecidable [19] and the universal language $(\Sigma \times \mathcal{D})^\omega$ is trivially NRA-definable.

3.3 Closure under composition

Closure under composition is a desirable property for transducers, which holds in the data-free setting [1]. We show that it also holds for functional NRT.

Theorem 9. *Let f, g be two functions defined by* NRT*. Then, their composition $f \circ g$ is (effectively) definable by some* NRT*.*

Sketch. We first give a sketch of the proof. Then, the rest of this section is devoted to the details of the proof.

By $f \circ g$ we mean $f \circ g : x \mapsto f(g(x))$. Assume f and g are defined by $T_f = (Q_f, R_f, q_0, F_f, \Delta_f)$ and $T_g = (Q_g, R_g, p_0, F_g, \Delta_g)$ respectively. Wlog we assume that the input and output finite alphabets of T_f and T_g are all equal to Σ, and that R_f and R_g are disjoint. We construct T such that $[\![T]\!] = f \circ g$. The proof is similar to the data-free case where the composition is shown via a product construction which simulates both transducers in parallel, executing the second on the output of the first. Assume T_g has some transition $p \xrightarrow{\sigma,\phi|\{r\},o} q$ where $o \in (\Sigma \times R_g)^*$. Then T has to be able to execute transitions of T_f while processing o, even though o does not contain any concrete data values (it is here the main important difference with the data-free setting). However, if T knows the equality types between R_f and R_g, then it is able to trigger the transitions of T_f. For example, assume that $o = (a, r_g)$ and assume that the content of r_g is equal to the content of r_f, r_f being a register of T_f, then if T_f has some transition of the form $p' \xrightarrow{a,r_f^{=}|\{r'_f\},o'} q'$ then T can trigger the transition $(p, q) \xrightarrow{\sigma,\phi|\{r\}\cup\{r'_f:=r_g\},o'} (p', q')$ where the operation $r'_f := r_g$ is a syntactic sugar on top of NRT that intuitively means "put the content of r_g into r'_f". In the following, we show that it is indeed syntactic sugar (Theorem 10) and provide more details on the construction. □

3.3.1 NRT with reassignments

An NRT *with reassignments* is defined as an NRT where operations on registers are of the form $r := curr$ to assign the current data read, or $r := r'$ to assign to r the value of r'. It is required that each register r appears at most once as the left-hand-side of an assignment $r := curr$ or $r := r'$. Given a configuration (q, τ) and some transition realising a set of instructions I and going to some state q', the new register configuration τ' is defined as follows. First, instructions $r := curr$ are executed, giving a new register configuration $\tau''(s) = \tau(s)$ if s has not been assigned the current data, and $\tau''(s) = d$ otherwise, if d is the current data. Then, instructions $r := r'$ are executed, so we let $\tau'(r) = \tau''(r')$ whenever there is an instruction $r := r'$, otherwise $\tau'(r) = \tau''(r)$.

We show in Theorem 10 that this feature does not add expressive power to the model.

3.3.2 NRT with explicit tests

First, to ease the construction, we need to explicit the tests, showing the equivalence with register automata as defined in [22]. Indeed, in our setting, they are represented using logical formulas, which allows for a more compact representation, but prevents us from keeping track of the equality relations between the different registers, which will be needed in the construction. For instance, after a transition $q \xrightarrow{a, r_1 \vee r_2 | \{r_3\}, r_1} q'$ is taken, we do not know whether we have $r_1 = r_3$ or $r_2 = r_3$, since it depends whether the current data is equal to the content of r_1 or of r_2. To explicit these two cases, such transition can thus be split into two distinct transitions, $q \xrightarrow{a, r_1 | \{r_3\}, r_1} q'$ and $q \xrightarrow{a, r_2 | \{r_3\}, r_1} q'$. Moreover, we also ignore whether such data is equal to some content of some other register r_4, which could entail additional equalities between registers. The two cases ($d = r_4$ and $d \neq r_4$) should thus yield two different transitions. Such operations can be generalised, allowing for all tests to be of the form $\phi_E = \bigwedge_{r \in E} r^= \wedge \bigwedge_{r \notin E} r^{\neq}$, where $E \subseteq R$ is a set of registers, meant to be *exactly* the set of registers to which the current data is equal. Thus, when a data is read, it satifies at most one test, and the equality relations between the registers can be updated.

Formally, a transition $q \xrightarrow{\sigma_{\mathtt{i}}, \phi | \sigma_{\mathtt{o}}, \mathsf{asgn}, r}_T q'$ is replaced by all the transitions $q \xrightarrow{\sigma_{\mathtt{i}}, \phi_E | \sigma_{\mathtt{o}}, \mathsf{asgn}, r}_T q'$ for all $E \subseteq R_k$ such that $\phi_E \Rightarrow \phi$ is true. Note that in general, such operation adds exponentially many transitions (but does not affect the number of states). In the rest of this section, we assume that all transducers are in this normal form, and simply write E for ϕ_E.

Now, we can move to the removal of reassignments:

Theorem 10. *For all* NRT *with reassignment T, one can construct an* NRT *T' such that $[\![T]\!] = [\![T']\!]$.*

Proof. The idea is to keep in memory (in the states of T') substitutions λ of registers by other registers, and define tests and outputs modulo those substitutions. We also add one extra register in T'. So, if R is the set of registers of T, we let $R' = R \cup \{s\}$ where s is some extra register. The states of T' maintain substitutions λ of type $R \to R'$, where the initial substitution λ_0 is the identity on R. Hence, the set of states of T' are pairs (q, λ) where q is a state of T'. Moreover, the reached substitution λ only depends on the sequence of transitions taken by T, so that we have a bijection between runs of T and that of T'. The transducer T' has the following invariant: for all runs of T over some finite prefix u reaching some configuration (q, τ), there exists a run of T' on u with the same sequence of T-states, reaching a configuration $((q, \lambda), \tau')$ such that for all $r \in R$, $\tau(r) = \tau'(\lambda(r))$. The converse also holds.

Formally, T' is constructed as follows: from any T-transition $q \xrightarrow[T]{\sigma, E|\mathsf{asgn}, o} q'$ we create the T'-transitions

$$(q, \lambda) \xrightarrow[T']{\sigma, E'|\{r_0\}, o'} (q', \lambda') \tag{1}$$

for all substitutions $\lambda : R \to R'$, such that the following conditions hold:

1. $E' = \{\lambda(r) \mid r \in E\}$
2. $r_0 \notin \lambda(R)$ (it exists since $|R'| > |R|$).
3. Let λ'' such that $\lambda''(r) = r_0$ if $(r := curr) \in \mathsf{asgn}$, and $\lambda''(r) = \lambda(r)$ otherwise. Then, for all $(r := r') \in \mathsf{asgn}$, we let $\lambda'(r) = \lambda''(r')$, and if r does not occur in some assignment $r := r'$, then we let $\lambda'(r) = \lambda''(r)$.
4. if $o = (\sigma_1, r_1) \dots (\sigma_k, r_k)$, then $o' = (\sigma_1, \lambda'(r_1)) \dots (\sigma_k, \lambda'(r_k))$.

Let us show that the invariant is preserved. It is obviously true for runs of length 0. Now, let r be a run of T on a word $u \in (\Sigma \times \mathcal{D})^*$, such that the last transition is $(q, \tau) \xrightarrow[T]{\sigma, E|\mathsf{asgn}, o} (q', \tau')$. By induction hypothesis there exists a run of T' on $u_1 \dots u_{n-1}$ reaching a configuration $((q, \lambda), \gamma)$ such that $\tau = \gamma \circ \lambda$. We show that we can extend this run with a T'-transition from (q, λ) as defined in Equation 1, towards a configuration $((q', \lambda'), \gamma')$ such that $\tau' = \gamma' \circ \lambda'$. Let $d \in \mathcal{D}$ the last data of u. We know that $\tau(r) = d$ for all $r \in E$. We have to show that $\gamma(r') = d$ for all $r' \in E'$, i.e. $\gamma(\lambda(r)) = d$ for all $r \in E$, by definition of E'. This is immediate as $\tau = \gamma \circ \lambda$. Hence, the test is satisfied and the T'-transition defined in Equation 1 can be triggered.

Now, let us show that $\tau' = \gamma' \circ \lambda'$. Let $r \in R$. First, note that $\gamma'(r) = \gamma(r)$ for all $r \neq r_0$. Then, let us consider several cases:

- if r has been untouched by asgn, then $\lambda'(r) = \lambda''(r) = \lambda(r)$ and $\tau'(r) = \tau(r)$, hence $\tau'(r) = \tau(r) = (\gamma \circ \lambda)(r) = (\gamma \circ \lambda')(r) = (\gamma' \circ \lambda')(r)$. The latter equality holds since $\gamma'(\lambda'(r)) = \gamma(\lambda'(r))$, because $\lambda'(r) \neq r_0$ (since r has been untouched by asgn).
- if $(r := curr) \in \mathsf{asgn}$, then $\lambda''(r) = r_0$ and since $(r := r') \notin \mathsf{asgn}$ for all $r' \in R$ (by our assumption that r only occurs at most once in the lhs of assignments), we also have $\lambda'(r) = \lambda''(r) = r_0$. Hence $\gamma' \circ \lambda'(r) = \gamma'(r_0) = d$ and $\tau'(r) = d$ since $(r := curr) \in \mathsf{asgn}$, hence $\tau'(r) = \gamma' \circ \lambda'(r)$.

- if $(r := r') \in \mathsf{asgn}$, then $\lambda'(r) = \lambda''(r')$. Then there are again several cases:

  1. $(r' := curr) \in \mathsf{asgn}$, then $\lambda''(r') = r_0$ and therefore $\gamma' \circ \lambda'(r) = \gamma'(r_0) = d$. Moreover, since T first executes $r' := curr$ and then $r := r'$, we get $\tau'(r) = \tau(r) = d$, so that $\gamma' \circ \lambda'(r) = \tau'(r)$
  2. $(r' := curr) \notin \mathsf{asgn}$, then $\lambda''(r') = \lambda(r')$. So, $\gamma' \circ \lambda(r) = \gamma' \circ \lambda(r')$. Necessarily, $\lambda(r') \neq r_0$ by definition of r_0, which must satisfy $r_0 \notin \lambda(R)$. Therefore, $\lambda(r')$ is not assigned the current data value by T', hence $\gamma'(\lambda(r')) = \gamma(\lambda(r'))$. Finally, we have:

$$\begin{aligned}
\gamma' \circ \lambda'(r) &= \gamma' \circ \lambda''(r') \\
&= \gamma' \circ \lambda(r') \\
&= \gamma \circ \lambda(r') && \text{since } \gamma'(\lambda(r')) = \gamma(\lambda(r')) \\
&= \tau(r') && \text{by induction hypothesis} \\
&= \tau'(r) && \text{since } (r := r') \in \mathsf{asgn}
\end{aligned}$$

So, we have shown that the invariant is preserved. We have left to show that $\gamma'(o') = \tau'(o)$, in the sense that if r is the ith register in o, then $\tau'(r) = \gamma'(r')$ where r' is the ith register of o'. By definition of o', we have $r' = \lambda'(r)$. By the invariant, $\tau'(r) = \gamma' \circ \lambda'(r)$, hence $\tau'(r) = \gamma'(r')$.

Overall, for all runs of T, there exists a run of T' simulating it in the sense that it follows the same sequence of T-states and produces the same output. This allows one to conclude that $[\![T]\!] \subseteq [\![T']\!]$. The other inclusion, $[\![T']\!] \subseteq [\![T]\!]$ also holds. It can be shown similarly by induction on the length of runs. If T' can trigger some transition, this transition has been obtained from a transition t of T, and one can show using the same equalities and reasoning as above that t can also be triggered and that the same invariant as above is satisfied, concluding the proof. ☐

3.3.3 Proof of Theorem 9, continued

In the following construction, we first assume that T_g produces at most one letter at a time, i.e., the output $o \in (\Sigma \times \mathcal{D})^*$ on all its transitions is such that $|o| = 1$. We explain later on how to generalise it to an arbitrary length. Under this assumption, we let $Q = Q_g \times Q_f \times 2^{R_g \times R_f}$ be the set of states of T and $R = R_g \uplus R_f$ its set of registers. Our construction satisfies the following invariant: if after reading a prefix x, T is in some configuration $((p, q, C), \tau)$, then T_g is in state p, T_f is in state q after reading the output of T_g on x, and $(r_g, r_f) \in C$ iff $\tau(r_g) = \tau(r_f)$. The *constraints* C are thus used to keep track of the equality relations between the registers of T_f and T_g, a standard technique in the setting of register automata. We now define the set of transitions. $(p, q, C) \xrightarrow{\sigma, E|\mathsf{asgn}, o} (p', q', C')$ if the following conditions hold:

1. there exists a transition $p \xrightarrow[T_g]{\sigma, E_g|\mathsf{asgn}_g, (\sigma', r_g)} p'$,

2. there exists a transition $q \xrightarrow[T_f]{\sigma', E_f|\mathsf{asgn}_f, o_f} q'$,

3. let C'' be the intermediate equality type defined as the equalities (r_1, r_2) when (r_1, r_2) were already equal and r_1 has not be reassigned the current data value, or r_1 has be reassigned the current data value, but this data value is equal to some $r_1' \in R_g$ (i.e. $r_1' \in E_g$) and r_1' was known to be equal to r_2. Formally:

$$\begin{aligned} C'' = &\{(r_1, r_2) \mid (r_1, r_2) \in C \wedge r_1 \notin \mathsf{asgn}_g\} \\ &\cup \{(r_1, r_2) \mid r_1 \in \mathsf{asgn}_g \wedge \exists r_1' \in E_g \cdot (r_1', r_2) \in C\} \end{aligned}$$

   Then, we require that for all $r_f \in R_f$, $r_f \in E_f$ iff $(r_g, r_f) \in C''$

4. $\mathsf{asgn} = \mathsf{asgn}_g \cup \{r := r_g \mid r \in \mathsf{asgn}_f\}$ (we assume that the instructions $r := r_g$ are executed after the assignments of the current data value)

5. $o = o_f$

6. $(r_1, r_2) \in C'$ iff $(r_1, r_2) \in C''$ and $r_2 \notin \mathsf{asgn}_f$, or $r_2 \in \mathsf{asgn}_f$ and $r_1 = r_g$

Finally, let us explain informally how to generalise this construction to outputs o of arbitrarily lengths. Assume that $o = (\sigma_1, r_g^1)(\sigma_2, r_g^2) \dots (\sigma_n, r_g^n)$. Note that n is bounded and depends only on T_f. The difference is at point 2 where instead we require the existence of a sequence of n transitions

$$q_0 \xrightarrow[T_f]{\sigma_1, E_f^1|\mathsf{asgn}_f^1, o_f^1} q_1 \xrightarrow[T_f]{\sigma_2, E_f^2|\mathsf{asgn}_f^2, o_f^2} q_2 \dots q_{n-1} \xrightarrow[T_f]{\sigma_n, E_f^n|\mathsf{asgn}_f^n, o_f^n} q_n$$

such that $q_0 = q$ and $q_n = q'$, and it satisfies some additional conditions that we now explain. First of all, starting from C'', since those transitions performs a series of assignments, the equality types between the registers of T_g and T_f may change along executing those transitions, which gives a series of equality types $C_1, \dots, C_n$ where C_i is the equality type before executing the ith transition (with $C_1 = C''$ as defined above). Then, we require that for all $r_f \in R_f$, $r_f \in E_f^i$ iff $(r_g^i, r_f) \in C_i$. From C_n and the last transition, we get a new equality type required to be equal to C'. We require that $\mathsf{asgn} = \mathsf{asgn}_g \cup \{r := r_g^i \mid r \in \mathsf{asgn}_f^i$ and step i is the last time r is assigned in the sequence of transitions$\}$. Regarding the output, we cannot just require that $o = o_f^1 \dots o_f^n$ as the values of the registers in T_f change along the sequence of transitions. However, for all i, the registers of T_f are necessarily equal to some register of T_g, i.e. for all $r \in R_f$, there exists $r' \in R_g$ such that $(r', r) \in C_i$. This is due to the fact that each assignment asgn_i assigns the value of r_g^i to the registers in asgn_i (assuming that the assignment of T_f are always non-empty, which can be ensured wlog, modulo adding some register that always receives the current data value). Hence, in the end we require that $o = \alpha(C_1, o_f^1) \dots \alpha(C_n, o_f^n)$ where $\alpha(C_i, o_f^i)$ substitutes in o_f^i any register $r_f \in R_f$ by some register $r \in R_g$ such that $(r, r_f) \in C_i$, concluding the proof. □

Remark 1. The proof of Theorem 9 does not use the hypothesis that f and g are functions, and actually shows a stronger result, namely that relations defined by NRT are closed under composition.

4 Computability and Continuity

We equip the set of (finite or infinite) data words with the usual distance: for $u, v \in (\Sigma \times \mathcal{D})^\omega$, $d(u, v) = 0$ if $u = v$ and $d(u, v) = 2^{-\|u \wedge v\|}$ otherwise. A sequence of (finite or infinite) data words $(x_n)_{n\in\mathbb{N}}$ converges to some infinite data word x if for all $\epsilon > 0$, there exists $N \geq 0$ such that for all $n \geq N$, $d(x_n, x) \leq \epsilon$.

In order to reason with computability, we assume in the sequel that the infinite set of data values $\mathcal{D}$ we are dealing with has an effective representation. For instance, this is the case when $\mathcal{D} = \mathbb{N}$.

We now define how a Turing machine can compute a function of data words. We consider deterministic Turing machines, which three tapes: a read-only one-way input tape (containing the infinite input data word), a two-way working tape, and a write-only one-way output tape (on which it writes the infinite output data word). Consider some input data word $x \in (\Sigma \times \mathcal{D})^\omega$. For any integer $k \in \mathbb{N}$, we let $M(x, k)$ denote the output written by M on its output tape after having read the k first cells of the input tape. Observe that as the output tape is write-only, the sequence of data words $(M(x, k))_{k\geq 0}$ is non-decreasing.

Definition 2 (Computability)**.** A function $f : (\Sigma \times \mathcal{D})^\omega \to (\Gamma \times \mathcal{D})^\omega$ is computable if there exists a deterministic multi-tape machine M such that for all $x \in \mathrm{dom}(f)$, the sequence $(M(x, k))_{k\geq 0}$ converges to $f(x)$.

Definition 3 (Continuity)**.** A function $f : (\Sigma \times \mathcal{D})^\omega \to (\Gamma \times \mathcal{D})^\omega$ is *continuous* at $x \in \mathrm{dom}(f)$ if (equivalently):

(a) for all sequences of data words $(x_n)_{n\in\mathbb{N}}$ converging towards x, where for all $i \in \mathbb{N}$, $x_i \in \mathrm{dom}(f)$, we have that $(f(x_n))_{n\in\mathbb{N}}$ converges to $f(x)$.

(b) $\forall i \geq 0, \exists j \geq 0, \forall y \in \mathrm{dom}(f), \|x \wedge y\| \geq j \Rightarrow \|f(x) \wedge f(y)\| \geq i$.

Then, f is continuous if and only if it is continuous at each $x \in \mathrm{dom}(f)$. Finally, a functional NRT T is *continuous* when $[\![T]\!]$ is continuous.

Example 3. We give an example of a non-continuous function f. The finite input and output alphabets are unary, and are therefore ignored in the description of f. Such function associates with every sequence $s = d_1 d_2 \cdots \in \mathcal{D}^\omega$ the word $f(s) = d_1^\omega$ if d_1 occurs infinitely many times in s, otherwise $f(s) = s$ itself.

The function f is not continuous. Indeed, by taking $d \neq d'$, the sequence of data words $d(d')^n d^\omega$ converges to $d(d')^\omega$, while $f(d(d')^n d^\omega) = d^\omega$ converges to $d^\omega \neq f(d(d')^\omega) = d(d')^\omega$.

Moreover, f is realisable by some NRT which non-deterministically guesses whether d_1 repeats infinitely many times or not. It needs only one register r

in which to store d_1. In the first case, it checks whether the current data d is equal the content r infinitely often, and in the second case, it checks that this test succeeds finitely many times, using Büchi conditions.

One can show that the register transducer T_{rename2} considered in Example 2 also realises a function which is not continuous, as the value stored in register r_2 may appear arbitrarily far in the input word. One could modify the specification to obtain a continuous function as follows. Instead of considering an infinite log, one considers now an infinite sequence of finite logs, separated by \$ symbols. The register transducer T_{rename3}, depicted in Figure 3, defines such a function.

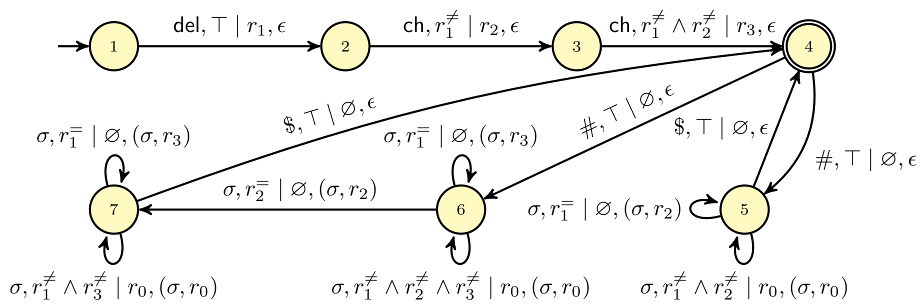

Figure 3: A register transducer T_{rename3}. This transducer is non-deterministic, yet it defines a continuous function.

We now prove the equivalence between continuity and computability for functions defined by NRT. One direction, namely the fact that computability implies continuity, is easy, almost by definition. For the other direction, we rely on the following lemma which states that it is decidable whether a word v can be safely output, only knowing a prefix u of the input. In particular, given a function f, we let $\hat{f}$ be the function defined over all finite prefixes u of words in $\mathrm{dom}(f)$ by $\hat{f}(u) = \bigwedge(f(uy) \mid uy \in \mathrm{dom}(f))$, the longest common prefix of all outputs of continuations of u by f. Then, we have the following decidability result:

Lemma 11. *The following problem is decidable. Given an* NRT T *defining a function* f*, two finite data words* $u \in (\Sigma \times \mathcal{D})^*$ *and* $v \in (\Gamma \times \mathcal{D})^*$*, decide whether* $v \preceq \hat{f}(u)$*.*

Proof. We decide the negation, i.e., whether there exists some continuation y such that $uy \in \mathrm{dom}(f)$ and $v \not\preceq f(uy)$. Since $f(uy)$ is infinite, it is equivalent to asking whether there is a mismatch between v and $f(uy)$ (a position i such that $v[i] \neq f(uy)[i]$). We reduce this test to the language emptiness test of some NRA A. The language defined by A is the set of data words $x_1 y_1 x_2 y_2 \ldots$ such that:

1. for all $i \geq 1$, $x_i \in \Sigma \times \mathcal{D}$, i.e. x_i is some input

2. for all $j \geq 1$, $y_j \in (\Gamma \times \mathcal{D})^*$, i.e. y_j is some output data word

3. $x_1 \dots x_k = u$ for some $k \geq 1$

4. $y_1 \dots y_\ell$ mismatches with v for some ℓ

5. there exists an accepting run of T of the form:

$$(q_0, \tau_0) \xrightarrow[T]{x_1|y_1} (q_1, \tau_1) \xrightarrow[T]{x_2|y_2} (q_2, \tau_2) \dots$$

Note that $L(A) \neq \varnothing$ iff $v \not\preceq \hat{f}(u)$.

It remains to show how to construct A. We construct A by taking the intersection of three NRA A_3, A_4 and A_5 which checks conditions $3, 4, 5$ (NRA are closed under language intersection). To construct A_3, since u is given, we can use as many registers as $|u|$ to store the data of u in some initial register configuration, used to check condition 3, by performing equality tests (the finite labels of u can be dealt with by storing them in the states of A_3). This automaton needs $O(|u|)$ states. We can proceed similarly for A_4, with the difference that at some point, A_4 has to guess the existence of some mismatch and perform some disequality test. Finally, A_5 just simulates T over $x_1 y_1 \dots$.

Since NRA emptiness is decidable [4] (even with some arbitrary initial configuration), this entails our result. ☐

Theorem 12. *Let f be a function defined by some* NRT *T. Then f is continuous iff f is computable.*

Proof. $\Leftarrow$ Assuming $f = [\![T]\!]$ is computable by some Turing machine M, we show that f is continuous. Indeed, consider some $x \in \mathrm{dom}(f)$, and some $i \geq 0$. As the sequence of finite words $(M(x,k))_{k \in \mathbb{N}}$ converges to $f(x)$ and these words have non-decreasing lengths, there exists $j \geq 0$ such that $|M(x,j)| \geq i$. Hence, for any data word $y \in \mathrm{dom}(f)$ such that $|x \wedge y| \geq j$, the behaviour of M on y is the same during the first j steps, as M is deterministic, and thus $|f(x) \wedge f(y)| \geq i$, showing that f is continuous at x.

$\Rightarrow$ Assume that f is continuous. We describe a Turing machine computing f; the corresponding algorithm is formalised as Algorithm 1. When reading a finite prefix $x[{:}j]$ of its input $x \in \mathrm{dom}(f)$, it computes the set P_j of all configurations (q, τ) reached by T on $x[{:}j]$. This set is updated along taking increasing values of j. It also keeps in memory the finite output word o_j that has been output so far. For any j, if $\mathsf{dt}(x[{:}j])$ denotes the data that appear in x, the algorithm then decides, for each input $(\sigma, d) \in \Sigma \times (\mathsf{dt}(x[{:}j]) \cup \{\mathsf{d}_0\})$ whether (σ, d) can safely be output, i.e., whether all accepting runs on words of the form $x[{:}j]y$, for an infinite word y, outputs at least $o_j(\sigma, d)$. The latter can be decided, given T, o_j and $x[{:}j]$, by Theorem 11. Note that it suffices to look at data in $\mathsf{dt}(x[{:}j]) \cup \{\mathsf{d}_0\}$ only since, by definition of NRT, any data that is output is necessarily stored in some register, and therefore appears in $x[{:}j]$ or is equal to d_0.

Let us show that M_f actually computes f. Let $x \in \mathrm{dom}(f)$. We have to show that the sequence $(M_f(x,j))_j$ converges to $f(x)$. Let o_j be the content of variable o of M_f when exiting the inner loop at line 8, when the outer loop (line

Algorithm 1: Algorithm describing the machine M_f computing f.

Data: $x \in \mathrm{dom}(f)$

1 $o := \epsilon$;

2 **for** $j = 0$ **to** ∞ **do**

3   **for** $(\sigma, d) \in \Sigma \times (\mathsf{dt}(x[:j]) \cup \{\mathsf{d}_0\})$ **do**

4   **if** $o.(\sigma, d) \preceq \hat{f}(x[:j])$ **then** `// such test is decidable by Theorem 11`

5             $o := o.(\sigma, d)$;

6   output (σ, d);

7   **end**

8   **end**

9 **end**

2) has been executed j times (hence j input symbols have been read). Note that $o_j = M_f(x, j)$. We have $o_1 \preceq o_2 \preceq \dots$ and $o_j \preceq \hat{f}(x[:j])$ for all $j \geq 0$. Hence, $o_j \preceq f(x)$ for all $j \geq 0$. To show that $(o_j)_j$ converges to $f(x)$, it remains to show that $(o_j)_j$ is non-stabilising, i.e. $o_{i_1} \prec o_{i_2} \prec \dots$ for some infinite subsequence $i_1 < i_2 < \dots$. First, note that f being continuous is equivalent to the sequence $(\hat{f}(x[:k]))_k$ converging to $f(x)$. Therefore we have that $f(x) \wedge \hat{f}(x[:k])$ can be arbitrarily long, for sufficiently large k. Let $j \geq 0$ and $(\sigma, d) = f(x)[|o_j|+1]$. By the latter property and the fact that $o_j.(\sigma, d) \preceq f(x)$, necessarily, there exists some $k > j$ such that $o_j.(\sigma, d) \preceq \hat{f}(x[:k])$. Moreover, by definition of NRT, d is necessarily a data that appears in some prefix of x, therefore there exists $k' \geq k$ such that d appears in $x[:k']$ and $o_j.(\sigma, d) \preceq \hat{f}(x[:k] \preceq \hat{f}(x[:k']$. This entails that $o_j.(\sigma, d) \preceq o_{k'}$. So, we have shown that for all for all j, there exists $k' > j$ such that $o_j \prec o_{k'}$, which concludes the proof. $\square$

Now that we have shown that computability is equivalent with continuity for functions defined by NRT, we exhibit a pattern which allows to decide continuity. Such pattern generalises the one of [3] to the setting of data words, the difficulty lying in showing that our pattern can be restricted to a finite number of data.

Theorem 13. *Let T be a functional NRT with k registers. Then, for all $X \subseteq \mathcal{D}$ such that $|X| \geq 2k+3$ and $\mathsf{d}_0 \in X$, T is not continuous at some $x \in (\Sigma \times \mathcal{D})^\omega$ if and only if T is not continuous at some $z \in (\Sigma \times X)^\omega$.*

Proof. The right-to-left direction is trivial. Now, let T be a functional NRT with k registers which is not continuous at some $x \in (\Sigma \times \mathcal{D})^\omega$. Let $f : \mathrm{dom}([\![T]\!]) \to (\Gamma \times \mathcal{D})^\omega$ be the function defined by T, as: for all $u \in \mathrm{dom}([\![T]\!])$, $f(u) = v$ where $v \in (\Gamma \times \mathcal{D})^\omega$ is the unique data word such that $(u, v) \in [\![T]\!]$.

Now, let $X \subseteq \mathcal{D}$ be such that $|X| \geq 2k+3$ and $\mathsf{d}_0 \in X$. We need to build two words u and v labelled over X which coincide on a sufficiently long prefix to allow for pumping, hence yielding a converging sequence of input data words whose images do not converge, witnessing non-continuity. To that end, we use a similar proof technique as for Theorem 6: we show that the language of interleaved runs whose inputs coincide on a sufficiently long prefix while their respective outputs mismatch before a given position is recognisable by an NRA, allowing us to use the indistinguishability property. We also ask that one run presents sufficiently many occurrences of a final state q_f, so that we can ensure that there exists a pair of configurations containing q_f which repeats in both runs.

On reading such u and v, the automaton behaves as a finite automaton, since the number of data is finite ([15, Proposition 1]). By analysing the respective runs, we can, using pumping arguments, bound the position on which the mismatch appears in u, then show the existence of a synchronised loop over u and v after such position, allowing us to build the sought witness for non-continuity.

Relabel over X Thus, assume T is not continuous at some point $x \in (\Sigma\times\mathcal{D})^\omega$. Let ρ be an accepting run of T over x, and let $q_f \in \inf(\mathsf{st}(\rho))\cap F$ be an accepting state repeating infinitely often in ρ. Then, let $i \geq 0$ be such that for all $j \geq 0$, there exists $y \in \mathrm{dom}(f)$ such that $\|x \wedge y\| \geq j$ but $\|f(x) \wedge f(y)\| \leq i$. Now, define $K = |Q| \times (2k+3)^{2k}$ and let $m = (2i+3) \times (K+1)$. Finally, pick j such that $\rho[1{:}j]$ contains at least m occurrences of q_f. Consider the language:

$$L = \big\{\rho_1 \otimes \rho_2 \big| \|\mathsf{in}(\rho_1) \wedge \mathsf{in}(\rho_2)\| \geq j, \|\mathsf{out}(\rho_1) \wedge \mathsf{out}(\rho_2)\| \leq i \text{ and}$$
$$\text{there are at least } m \text{ occurrences of } q_f \text{ in } \rho_1[1{:}j]\big\}$$

By Theorem 3, $L_\otimes(T)$ is recognised by an NRA with $2k$ registers. Additionnally, by Theorem 4, M^i_j is recognised by an NRA with 2 registers. Thus, $L = L_\otimes(T) \cap O^{q_f}_{m,j} \cap M^i_j$, where $O^{q_f}_{m,j}$ checks there are at least m occurrences of q_f in $\rho_1[1{:}j]$ (this is easily doable from the automaton recognising $L_\otimes(T)$ by adding an m-bounded counter), is recognisable by an NRA with $2k+2$ registers.

Choose $y \in \mathrm{dom}(f)$ such that $\|x \wedge y\| \geq j$ but $\|f(x) \wedge f(y)\| \leq i$. By letting ρ_1 (resp. ρ_2) be an accepting run of T over x (resp. y) we have $\rho_1 \otimes \rho_2 \in L$, so $L \neq \varnothing$. By Theorem 2, it means $L \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega) \neq \varnothing$. Let $w = \rho'_1 \otimes \rho'_2 \in L \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega)$, $u = \mathsf{in}(\rho'_1)$ and $v = \mathsf{in}(\rho'_2)$. Then, $\|u \wedge v\| \geq j$, $\|f(u) \wedge f(v)\| \leq i$ and there are at least m occurrences of q_f in $\rho_1[1{:}j]$.

Now, we depict ρ'_1 and ρ'_2 in Figure 4, where we decompose u as $u = u_1 \dots u_m \cdot s$ and v as $v = u_1 \dots u_m \cdot t$; their corresponding images being respectively $u' = u'_1 \dots u'_m \cdot s'$ and $u'' = u''_1 \dots u''_m t''$. We also let $l = (i+1)(K+1)$ and $l' = 2(i+1)(K+1)$. Since the data of u, v and w belong to X, we know that $\tau_i, \mu_i : R \to X$.

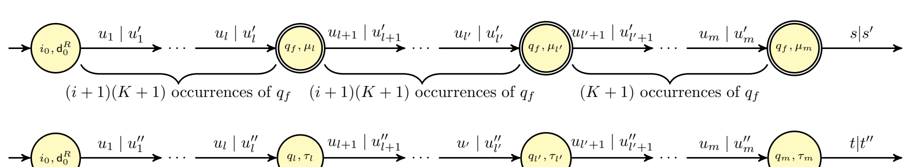

Figure 4: Runs of f over $u = u_1 \dots u_m \cdot s$ and $v = u_1 \dots u_m \cdot t$.

Repeating configurations First, let us observe that in a partial run of ρ'_1 containing more than $|Q|\times|X|^k$ occurrences of q_f, there is at least one productive transition, i.e. a transition whose output is $o \neq \varepsilon$. Otherwise, by the pigeonhole principle, there exists a configuration $\mu : R \to X$ such that (q_f, μ) occurs at least twice in the partial run. Since all transitions are improductive, it would mean that, by writing w the corresponding part of input, we have $(q_f, \mu) \xrightarrow[T]{w|\varepsilon} (q_f, \mu)$. This partial run is part of ρ'_1, so, in particular, (q_f, μ) is accessible, hence by taking w_0 such that $(i_0, \tau_0) \xrightarrow[T]{w_0|w'_0} (q_f, \mu)$, we have that $f(w_0 w^\omega) = w'_0$, which is a finite word, contradicting our assumption that all accepting runs produce an infinite output. This implies that, for any $n \geq |Q| \times |X|^k$ (in particular for $n = l$), $\|u'_1 \dots u'_n\| \geq i + 1$.

Locate the mismatch Again, upon reading $u_{l+1} \dots u_{l'}$, there are $(i+1)(K+1)$ occurrences of q_f. There are two cases:

(a) There are at least $i + 1$ productive transitions in ρ'_2. Then, we obtain that $\|u''_1 \dots u''_{l'}\| > i$, so $\mathsf{mismatch}(u'_1 \dots u'_{l'}, u''_1 \dots u''_{l'})$, since we know $\|f(u) \wedge f(v)\| \leq i$ and they are respectively prefixes of $f(u)$ and $f(v)$, both of length at least $i + 1$. Afterwards, upon reading $u_{l'+1} \dots u_m$, there are $K + 1 > |Q| \times |X|^{2k}$ occurrences of q_f, so, by the pigeonhole principle, there is a repeating pair: there exist indices p and p' such that $l' \leq p < p' \leq m$ and $(q_f, \mu_p) = (q_f, \mu_{p'})$, $(q_p, \tau_p) = (q_{p'}, \tau_{p'})$. Thus, let $z_P = u_1 \dots u_p$, $z_R = u_{p+1} \dots u_{p'}$ and $z_C = u_{p'+1} \dots u_m \cdot t$ (P stands for *prefix*, R for *repeat* and C for *continuation*; we use capital letters to avoid confusion with indices). By denoting $z'_P = u'_1 \dots u'_p$, $z'_R = u'_{p+1} \dots u'_{p'}$, $z''_P = u''_1 \dots u''_p$, $z''_R = u''_{p+1} \dots u''_{p'}$ and $z''_C = u''_{p'+1} \dots u''_m \cdot t''$ the corresponding images, $z = z_P \cdot z_R{}^\omega$ is a point of discontinuity. Indeed, define $(z_n)_{n\in\mathbb{N}}$ as, for all $n \in \mathbb{N}$, $z_n = z_P \cdot z_R^n \cdot z_C$. Then, $(z_n)_{n\in\mathbb{N}}$ converges towards z, but, since for all $n \in \mathbb{N}$, $f(z_n) = z''_P \cdot z''_L{}^n \cdot z''_C$, we have that $f(z_n) \underset{n\infty}{\not\longrightarrow} f(z) = z'_P \cdot z'_L{}^\omega$, since $\mathsf{mismatch}(z'_P, z''_P)$.

(b) Otherwise, by the same reasoning as above, it means there exists a repeating pair with only improductive transitions in between: there exist indices p and p' such that $l \leq p < p' \leq l'$, $(q_f, \mu_p) = (q_f, \mu_{p'})$, $(q_p, \tau_p) = (q_{p'}, \tau_{p'})$, and $(q_f, \mu_p) \xrightarrow{u_{p+1} \dots u_{p'}|\varepsilon} (q_f, \mu_{p'})$, $(q_p, \tau_p) \xrightarrow{u_{p+1} \dots u_{p'}|\varepsilon} (q_{p'}, \tau_{p'})$. Then, by taking $z_P = u_1 \dots u_p$, $z_R = u_{p+1} \dots u_{p'}$ and $z_C = u_{p'+1} \dots u_m \cdot t$, we have, by letting $z'_P = u'_1 \dots u'_p$, $z'_R = u'_{p+1} \dots u'_{p'}$, $z''_P = u''_1 \dots u''_p$, $z''_R = \varepsilon$ and $z''_C = u''_{p'+1} \dots u''_m \cdot t''$, that $z = z_P \cdot z_R{}^\omega$ is a point of discontinuity. Indeed, define $(z_n)_{n\in\mathbb{N}}$ as, for all $n \in \mathbb{N}$, $z_n = z_P \cdot z_R^n \cdot z_C$. Then, $(z_n)_{n\in\mathbb{N}}$ indeed converges towards z, but, since for all $n \in \mathbb{N}$, $f(z_n) = z''_P \cdot z''_C$, we have that $f(z_n) \underset{n\infty}{\not\longrightarrow} f(z) = z'_P \cdot z'_R{}^\omega$, since $\mathsf{mismatch}(z'_P, z''_P \cdot z''_C)$ (the mismatch necessarily lies in z'_P, since $\|z'_P\| \geq i + 1$). □

Corollary 14. *Deciding whether an* NRT *defines a continuous function is* PSPACE-*complete.*

Proof. Let $X \subseteq \mathcal{D}$ be a set of size $2k+3$ containing d_0. By Theorem 13, T is not continuous iff it is not continuous at some $z \in (\Sigma \times X)^\omega$, iff $[\![T]\!] \cap \big((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega\big)$ is not continuous. By Theorem 1, such relation is recognisable by a finite transducer T_X with $O(|Q| \times |X|^{|R|})$ states, which can be built on-the-fly. By [3], the continuity of functions defined by NFT is decidable in NLogSpace, which yields a PSpace procedure.

For the hardness, we reduce again from the emptiness problem of register automata, which is PSpace-complete [4]. Let A be a register automaton over some alphabet $\Sigma \times \mathcal{D}$. We construct a transducer T which defines a continuous function iff $L(A) = \varnothing$ iff the domain of T is empty. Let f be a non-continous function realised by some NRT H (it exists by Example 3). Then, let $\# \notin \Sigma$ be a fresh symbol, and define the function g as the function mapping any data word of the form $w(\#, d)w'$ to $w(\#, d)f(w')$ if $w \in L(A)$. The function g is realised by an NRT which simulates A and copies its inputs on the output to implement the identity, until it sees $\#$. If it was in some accepting state of A before seeing $\#$, it branches to some initial state of H and proceeds executing H. If there is some $w_0 \in L(A)$, then the subfunction g_{w_0} mapping words of the form $w_0(\#, d)w'$ to $w_0(\#, d)f(w')$ is not continuous, since f is not. Hence g is not continuous. Conversely, if $L(A) = \varnothing$, then $\mathrm{dom}(g) = \varnothing$, so g is continuous. $\square$

In [3], non-continuity is characterised by a specific pattern (Lemma 21, Figure 1), i.e. the existence of some particular sequence of transitions. By applying this characterisation to the finite transducer recognising $[\![T]\!] \cap ((\Sigma \times X)^\omega \times (\Gamma \times X)^\omega)$, as constructed in Theorem 1, we can characterise non-continuity by a similar pattern, which will prove useful to decide (non-)continuity of test-free NRT in NLogSpace (cf Section 5):

Corollary 15 ([3])**.** *Let T be an NRT with k registers. Then, for all $X \subseteq \mathcal{D}$ such that $|X| \geq 2k+3$ and $\mathsf{d}_0 \in X$, T is not continuous at some $x \in (\Sigma \times \mathcal{D})^\omega$ if and only if it has the pattern of Figure 5.*

$\exists u, v \in (\Sigma \times X)^*$,
$\exists u', u'', v', v'' \in (\Gamma \times X)^*$,
$\exists w \in (\Sigma \times X)^\omega$,
$\exists w'' \in (\Gamma \times X)^\omega$,
$\exists q \in Q, \exists q_f \in F$,
$\exists \mu, \tau : R \to X$,
$\mathsf{mismatch}(u', u'') \vee$
$v'' = \varepsilon \wedge \mathsf{mismatch}(u', u''w'')$

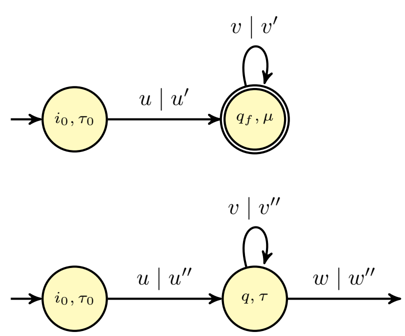

Figure 5: A pattern characterising non-continuity of functions definable by an NRT

5 Test-free Register Transducers

In [7], we introduced a restriction which allows to recover decidability of the bounded synthesis problem for specifications expressed as non-deterministic register automata. Applied to transducers, such restriction also yields polynomial complexities when considering the functionality and computability problems.

An NRT T is *test-free* when its transition function does not depend on the tests conducted over the input data. Formally, we say that T is *test-free* if for all transitions $q \xrightarrow[T]{\sigma,\phi|\mathsf{asgn},o} q'$ we have $\phi = \top$. Thus, we can omit the tests altogether and its transition relation can be represented as $\Delta' \subseteq Q \times \Sigma \times 2^R \times (\Gamma \times R)^* \times Q$.

Example 4. Consider the function $f : (\Sigma \times \mathcal{D})^\omega \to (\Gamma \times \mathcal{D})^\omega$ associating, to $x = (\sigma_1, d_1)(\sigma_2, d_2)\ldots$, the value $(\sigma_1, d_1)(\sigma_2, d_1)(\sigma_3, d_1)\ldots$ if there are infinitely many a in x, and $(\sigma_1, d_2)(\sigma_2, d_2)(\sigma_3, d_2)\ldots$ otherwise.

f can be implemented using a test-free NRT with one register: it initially guesses whether there are infinitely many a in x, if it is the case, it stores d_1 in the single register r, otherwise it waits for the next input to get d_2 and stores it in r. Then, it outputs the content of r along with each σ_i. f is not continuous, as even outputting the first data requires reading an infinite prefix when $d_1 \neq d_2$.

Note that when a transducer is test-free, the existence of an accepting run over a given input x only depends on its finite labels. Hence, the existence of two outputs y and z which mismatch over data can be characterised by a simple pattern (Figure 6), which allows to decide functionality in polynomial time:

Theorem 16. *Deciding whether a test-free* NRT *is functional is in* PTIME.

Proof. Let T be a test-free NRT such that T is not functional. Then, there exists $x \in (\Sigma \times \mathcal{D})^\omega$, $y, z \in (\Gamma \times \mathcal{D})^\omega$ such that $(x, y), (x, z) \in [\![T]\!]$ and $y \neq z$. Then, let i be such that $y[i] \neq z[i]$. There are two cases. Either $\mathsf{lab}(y[i]) \neq \mathsf{lab}(z[i])$, which means that the finite transducer T' obtained by ignoring the registers of T is not functional. By Theorem 5, such property can be decided in NLOGSPACE, so let us focus on the second case: $\mathsf{dt}(y[i]) \neq \mathsf{dt}(z[i])$.

We first give an outline of the proof: observe that an input x admits two outputs which mismatch over data if and only if it admits two runs which respectively store $x[j]$ and $x[j']$ such that $x[j] \neq x[j']$ and output them later at the same output position i; the outputs y and z are then such that $\mathsf{dt}(y[i]) \neq \mathsf{dt}(z[i])$. Since T is test-free, the existence of two runs over the same input x only depends on its finite labels. Then, the registers containing respectively $x[j]$ and $x[j']$ should not be reassigned before being output, and should indeed output their content at the same position i (cf Figure 6). Besides, again because of test-freeness, we can always assume that x is such that $x[j] \neq x[j']$. Overall, such pattern can be checked by a 2-counter Parikh automaton, whose emptiness is decidable in PTIME [8] (under conditions that are satisfied here).

Now, we move to the detailed proof. The case where the mismatch is over the labels reduces to deciding the functionality of an NRT, which is in NLOGSPACE by Theorem 5. Let us thus move the case where the mismatch occurs over the data.

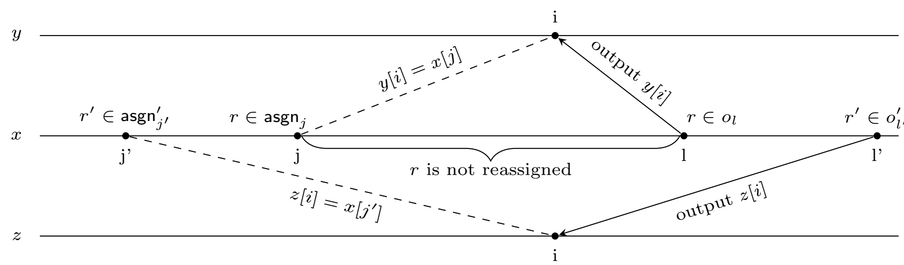

Figure 6: A situation characterising the existence of a mismatch over data. Since acceptance does not depend on data, we can always choose x such that $\mathsf{dt}(x[j]) \neq \mathsf{dt}(x[j'])$. Here, we assume that the labels of x, y and z range over a unary alphabet; in particular $y[i] = x[j]$ iff $\mathsf{dt}(y[i]) = \mathsf{dt}(x[j])$. Finally, for readability, we did not write that r' should not be reassigned between j' and l'. Note that the position of i with regards to j, j', l and l' does not matter; nor does the position of l w.r.t. l'.

Characterising a Mismatch Let us show that there exists an input $x \in (\Sigma \times \mathcal{D})^\omega$, two outputs $y, z \in (\Gamma \times \mathcal{D})^\omega$ such that $\mathsf{dt}(y) \neq \mathsf{dt}(z)$ if and only if there exists two finite sequences of transitions $\nu : q_0 \xrightarrow{\sigma_1 | \mathsf{asgn}_1, o_1} \dots \xrightarrow{\sigma_n | \mathsf{asgn}_n, o_n} q_n$ and $\nu' : q'_0 \xrightarrow{\sigma'_1 | \mathsf{asgn}'_1, o'_1} \dots \xrightarrow{\sigma'_n | \mathsf{asgn}'_n, o'_n} q'_n$, two registers r and r', an (output) position i and (input) positions $j \neq j'$, $n \geq l \geq j$ and $n \geq l' \geq j'$ such that:

1. The input labels are the same: for all $1 \leq k \leq n, \sigma_k = \sigma'_k$
2. ν and ν' can yield accepting runs: q_n and q'_n are coaccessible with the same data word
3. l and l' are the respective input positions at which the output data at position i is produced: $\|o_1 \dots o_{l-1}\| < i \leq \|o_1 \dots o_l\|$ and $\|o'_1 \dots o'_{l-1}\| < i \leq \|o'_1 \dots o'_l\|$
4. r and r' are the registers whose content is output at that moment: by denoting $w = o_1 \cdot o_2 \dots$ and $w' = o'_1 \cdot o'_2 \dots$, we have that $w[i] = (\sigma_i, r)$ and $w'[i] = (\sigma'_i, r')$
5. r and r' are assigned at a different position, respectively j and j': $r \in \mathsf{asgn}_j$, $r' \in \mathsf{asgn}_{j'}$
6. r and r' are not reassigned before being output: $\forall j < k \leq l, r \notin \mathsf{asgn}_k$, $\forall j' < k \leq l', r \notin \mathsf{asgn}'_k$; in other words, j and j' are the respective *origins* of the output at position i over ρ and ρ', in the sense of [7]

$\Leftarrow$ Choose $x = (\sigma_1, d_1) \dots (\sigma_n, d_n)(\sigma_{n+1}, d_{n+1}) \dots$, such that $d_j \neq d_{j'}$. Then, since T is test-free, by items 1 and 2, ν and ν' both yield accepting runs of T, that we respectively denote ρ and ρ'. We then let $y = \mathsf{out}(\rho)$ and $z = \mathsf{out}(\rho')$. Now, let (q_j, τ_j) be the configuration of T at position j. Since $r \in \mathsf{asgn}_j$, we have that $\tau_j(r) = d_j$. Then, by item 6, since for all $j < k \leq l$, $r \notin \mathsf{asgn}_k$, we get that $\tau_l(r) = d_j$. Then, by items 3 and 4, we get that $\mathsf{dt}(y[i]) = d_j$. Similarly, we have that $\mathsf{dt}(z[i]) = d'_j$. Overall, we get that x, y and z are such that $(x, y), (x, z) \in [\![T]\!]$ and $\mathsf{dt}(y) \neq \mathsf{dt}(z)$.

$\Rightarrow$ Now, let $x \in (\Sigma \times \mathcal{D})^\omega$, $y, z \in (\Gamma \times \mathcal{D})^\omega$ be such that $(x, y), (x, z) \in [\![T]\!]$ and $\mathsf{dt}(y) \neq \mathsf{dt}(z)$. Let $i \in \mathbb{N}$ be such that $\mathsf{dt}(y[i]) \neq \mathsf{dt}(z[i])$. Let ρ and ρ' be two accepting runs of T such that $\mathsf{in}(\rho) = \mathsf{in}(\rho') = x$ and $\mathsf{out}(\rho) = y$, $\mathsf{out}(\rho') = z$. Then, let l and l' be such that $\|o_1 \dots o_{l-1}\| < i \leq \|o_1 \dots o_l\|$ and $\|o'_1 \dots o'_{l-1}\| < i \leq \|o'_1 \dots o'_l\|$, and define $n = \max(l, l')$. Let $\nu = q_0 \xrightarrow{\sigma_1 | \mathsf{asgn}_1, o_1} q_1 \dots \xrightarrow{\sigma_n | \mathsf{asgn}_n, o_n} q_n$ and $\nu' = q'_0 \xrightarrow{\sigma'_1 | \mathsf{asgn}'_1, o'_1} q'_1 \dots \xrightarrow{\sigma'_n | \mathsf{asgn}'_n, o'_n} q_n$ be their respective sequences of transitions, truncated at length n.

First, for all $1 \leq k \leq n, \sigma_k = \sigma'_k = x[k]$, and $\sigma_{n+1}\sigma_{n+2}\dots$ indeed yiels a final run from q_n and q'_n since ρ and ρ' are accepting. Now, let r and r' be such that, by denoting $w = o_1 \cdot o_2 \dots$ and $w' = o'_1 \cdot o'_2 \dots$, we have $w[i] = (\sigma_i, r)$ and $w'[i] = (\sigma'_i, r')$. Then, define $j = \max\{k \leq l \mid r \in \mathsf{asgn}_k\}$ and $j' = \max\{k' \leq l' \mid r' \in \mathsf{asgn}_{k'}\}$ (with the convention that $\max \varnothing = 0$). By definition, item 6 then holds and $l \geq j$, $l' \geq j'$. Finally, by denoting $d_j = \mathsf{dt}(x[j])$ and $d_{j'} = \mathsf{dt}(x[j'])$ (with the convention that $\mathsf{dt}(x[0]) = \mathsf{d}_0$), we have that $\tau_j(r) = d_j$ and $\tau_{j'}(r') = d_{j'}$. Since r and r' are not reassigned before position l and l' respectively, we get that $\tau_l(r) = d_j$ and $\tau_{l'}(r') = d_{j'}$, so $d_j = \mathsf{dt}(y[i])$ and $d_{j'} = \mathsf{dt}(z[i])$, which means that $d_j \neq d_{j'}$, so $j \neq j'$.

Recognising the Pattern Now, checking those properties can be done by a 2-counter Parikh automaton. A Parikh automaton is a finite automaton equipped with a finite number of counters that it can increment and decrement, and which take their values in $\mathbb{Z}$ (i.e. they are allowed to go below zero). There are no tests over the counters during the run, but, to determine acceptance, at the end of the run, the automaton checks whether its counters satisfy a given Presburger formula, or, equivalently, belong to some semi-linear set. When increments and decrements are encoded in unary, the semi-linear set is given explicitly, and the dimension (i.e. the number of counters) is fixed, emptiness of Parikh automata is in PTime. This is the case here: there are only increments, which can be given in unary as they correspond to the length of the output words in the transitions, and at the end of the run, we only need to check that the two counters are equal, i.e. belong to the diagonal $c_1 = c_2$.

Let us describe the automaton: initially, it guesses the finite labels of ν and ν'. Then, there are two phases: it first simulates the two partial runs in parallel, only with regards to input labels. Formally, in this phase it can take a transition $q \xrightarrow{\sigma} q'$ if and only if there exists some transition $q \xrightarrow{\sigma | \mathsf{asgn}, o} q'$ in T. All the while, it keeps track of the length of the output which has been produced by each run using its two counters c_1 and c_2: c_1 is each time incremented by $\|o_1\|$ (resp. c_2 by $\|o_2\|$). It then guesses position j and j', and the corresponding registers r and r'. Afterwards, it keeps simulating both runs, but additionnally checks that

r (resp. r') is not reassigned after position j (resp. j'). To that end, it avoids all transitions $q \xrightarrow{\sigma|\mathsf{asgn},o} q'$ such that $\mathsf{asgn} \ni r$ (resp. $\mathsf{asgn} \ni r'$). Then, it guesses positions l and l' which are the respective origins of the mismatch. At this point, it increments c_1 (resp. c_2) by $\|w\|$, where w is such that $o_l = w_1 r w_1'$ (resp. $\|w_2\|$, where w_2 is such that $o_{l'} = w_2 r w_2'$) and then stops counting the length of the corresponding outputs until the end of the run, i.e. it stops incrementing c_1 (resp. c_2). Finally, at the end of the run, it checks that $c_1 = c_2$, and that q_n and q_n' are co-accessible with the same ω-word (here, the registers do not matter anymore). This is doable in polynomial time: precompute all pairs (q_n, q_n') such that $\mathsf{coacc}(q_n, q_n')$ hold, such predicate being computable in NLogSpace as demonstrated in Theorem 5.

Then, such automaton is polynomial in the size of T. Since emptiness of such automaton is in PTime, we can decide the above properties, i.e. whether T is functional, in PTime. ☐

Now, let us move to the case of continuity. Here again, the fact that test-free NRT conduct no test over the input data allows to focus on the only two registers that are responsible for the mismatch, the existence of an accepting run being only determined by finite labels.

Theorem 17. *Deciding whether a test-free* NRT *defines a continuous function is in* PTime.

Proof. Let T be a test-free NRT.

A simpler pattern Let us first show that T is continuous if and only if T has the pattern of Figure 7, where r is coaccessible (since acceptance only depends on finite labels, T can be trimmed[1] in polynomial time). By Theorem 13, we

$\exists u, v \in (\Sigma \times \mathcal{D})^*$,
$\exists u', u'', v', v'' \in (\Gamma \times \mathcal{D})^*$,
$\exists z \in (\Sigma \times \mathcal{D})^\omega$,
$\exists z'' \in (\Gamma \times \mathcal{D})^\omega$,
$\exists q, r \in Q, \exists q_f \in F$,
$\mathsf{mismatch}(u', u'') \vee$
$v'' = \varepsilon \wedge \mathsf{mismatch}(u', u''z'')$

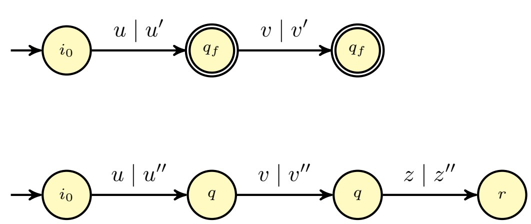

Figure 7: A pattern characterising non-continuity of functions definable by an NRT. Configurations are not depicted, as there are no conditions on them. We unrolled the loops to highlight the fact that they do not necessarily loop back to the same configuration.

already know that T is not continuous if and only if it has the pattern of Figure 5.

[1]We say that T is trim when all its states are both accessible and coaccessible.

Now, the left-to-right direction is quite direct, since the pattern of Figure 7 is simpler: assume T is not continuous. Then it has the pattern of Figure 5. Thus, if $\mathsf{mismatch}(u', u'')$, by picking $w = w'' = \varepsilon$ and $r = q$, we get the pattern of Figure 7. Otherwise, choose two finite data words z and z'' and r such that $z \prec w$, $z'' \prec w''$, $q \xrightarrow{z|z''} q'$ and $\mathsf{mismatch}(u', u''z'')$. Such finite words exist because the mismatch between u' and $u''w''$ happens at some finite position, and it suffices to truncate w and w'' up to such position.

Conversely, assume T has the pattern of Figure 7. Then, let $x = uv^\omega$, and define $(y_n)_{n\in\mathbb{N}}$ as $y_n = uv^n wz$ for all $n \in \mathbb{N}$, where $z \in (\Sigma \times \omega)$ is such that there is a final run over z from r. Such z exists since r is coaccessible; in the following we denote z'' its corresponding image. Then, $y_n \xrightarrow[n\to\infty]{} x$; however, for all $n \in \mathbb{N}$, $f(y_n) = u''(v'')^n w'' z''$, so $\|f(x) \wedge f(y_n)\| \leq \|u'\|$, since either $v'' \neq \varepsilon$ and then there is a mismatch between u' and u'', or there is a mismatch between u' and $u''w''$, which means that T is not continuous.

Checking the pattern Now, it remains to show that such simpler pattern can be checked in PTime. We treat each part of the disjunction separately:

(a) there exists u, u', u'', v, v', v'' such that $i_0 \xrightarrow{u|u'} q_f \xrightarrow{v|v'} q_f$ and $i_0 \xrightarrow{u|u''} q \xrightarrow{v|v''} q$, where $q_f \in F$ and $\mathsf{mismatch}(u', u'')$. Then, as shown in the proof of Theorem 16, there exists a mismatch between some u' and u'' produced by the same input u if and only if there exists two runs and two registers r and r' assigned at two distinct positions, and later on output at the same position. Such pattern can be checked by a 2-counter Parikh automaton similar to the one described in the proof of Theorem 16; the only difference is that here, instead of checking that the two end states are coaccessible with a common ω-word, we only need to check that $q_f \in F$ and that there is a synchronised loop over q_f and q, which are regular properties that can be checked by the Parikh automaton with only a polynomial increase.

(b) there exists $u, u', u'', v, v', z, z''$ such that $i_0 \xrightarrow{u|u'} q_f \xrightarrow{v|v'} q_f$ and $i_0 \xrightarrow{u|u''} q \xrightarrow{v|\varepsilon} q \xrightarrow{z|z''} r$, where $q_f \in F$ and $\mathsf{mismatch}(u', u''z'')$. By examining again the proof of Theorem 16, it can be shown that to obtain a mismatch, it suffices that the input is the same for both runs only up to position $\max(j, j')$. More precisely, there is a mismatch between u' and $u''z''$ if and only if there exists two registers r and r' and two positions $j, j' \in \{1, \dots, \|u\|\}$ such that $j \neq j'$, r is stored at position j, r' is stored at position j', r and r' are respectively output at input positions $l \in \{1, \dots, \|u\|\}$ and $l' \in \{1, \dots, \|uz\|\}$ and they are not reassigned in the meantime. Again, such property, along with the fact that $q_f \in F$ and the existence of a synchronised loop can be checked by a 2-counter Parikh automaton of polynomial size.

Overall, deciding whether a test-free NRT is continuous is in PTime. $\square$

References

[1] Berstel, J.: Transductions and Context-free Languages. Teubner Verlag (1979), http://www-igm.univ-mlv.fr/~berstel/LivreTransductions/LivreTransductions.html

[2] Carayol, A., Löding, C.: Uniformization in Automata Theory. In: Proceedings of the 14th Congress of Logic, Methodology and Philosophy of Science Nancy, July 19-26, 2011. pp. 153–178. London: College Publications (2014)

[3] Dave, V., Filiot, E., Krishna, S.N., Lhote, N.: Deciding the computability of regular functions over infinite words. CoRR **abs/1906.04199** (2019), http://arxiv.org/abs/1906.04199

[4] Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. **10**(3), 16:1–16:30 (2009). https://doi.org/10.1145/1507244.1507246, https://doi.org/10.1145/1507244.1507246

[5] Durand-Gasselin, A., Habermehl, P.: Regular transformations of data words through origin information. In: Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. pp. 285–300 (2016)

[6] Ehlers, R., Seshia, S.A., Kress-Gazit, H.: Synthesis with identifiers. In: Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 8318. pp. 415–433. VMCAI 2014 (2014)

[7] Exibard, L., Filiot, E., Reynier, P.: Synthesis of data word transducers. In: 30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands. pp. 24:1–24:15 (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.24, https://doi.org/10.4230/LIPIcs.CONCUR.2019.24

[8] Figueira, D., Libkin, L.: Path logics for querying graphs: Combining expressiveness and efficiency. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. pp. 329–340 (2015). https://doi.org/10.1109/LICS.2015.39, https://doi.org/10.1109/LICS.2015.39

[9] Filiot, E., Jecker, I., Löding, C., Winter, S.: On equivalence and uniformisation problems for finite transducers. In: 43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, Rome, Italy. pp. 125:1–125:14 (2016), https://doi.org/10.4230/LIPIcs.ICALP.2016.125

[10] Filiot, E., Mazzocchi, N., Raskin, J.: A pattern logic for automata with outputs. In: Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings. pp. 304–317 (2018)

[11] Gire, F.: Two decidability problems for infinite words. Inf. Process. Lett. **22**(3), 135–140 (1986). https://doi.org/10.1016/0020-0190(86)90058-X, `https://doi.org/10.1016/0020-0190(86)90058-X`

[12] Holtmann, M., Kaiser, L., Thomas, W.: Degrees of lookahead in regular infinite games. Logical Methods in Computer Science **8**(3) (2012). https://doi.org/10.2168/LMCS-8(3:24)2012, `https://doi.org/10.2168/LMCS-8(3:24)2012`

[13] II, K.C., Pachl, J.K.: Equivalence problems for mappings on infinite strings. Information and Control **49**(1), 52–63 (1981). https://doi.org/10.1016/S0019-9958(81)90444-7, `https://doi.org/10.1016/S0019-9958(81)90444-7`

[14] J.R. Büchi, L.H. Landweber: Solving sequential conditions finite-state strategies. Transactions of the American Mathematical Society **138**, 295–311 (1969)

[15] Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (Nov 1994). https://doi.org/10.1016/0304-3975(94)90242-9, `http://dx.doi.org/10.1016/0304-3975(94)90242-9`

[16] Khalimov, A., Kupferman, O.: Register-bounded synthesis. In: 30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands. pp. 25:1–25:16 (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.25, `https://doi.org/10.4230/LIPIcs.CONCUR.2019.25`

[17] Khalimov, A., Maderbacher, B., Bloem, R.: Bounded synthesis of register transducers. In: Automated Technology for Verification and Analysis, 16th International Symposium, ATVA 2018, Los Angeles, October 7-10, 2018. Proceedings (2018)

[18] Libkin, L., Tan, T., Vrgoc, D.: Regular expressions for data words. J. Comput. Syst. Sci. **81**(7), 1278–1297 (2015). https://doi.org/10.1016/j.jcss.2015.03.005, `https://doi.org/10.1016/j.jcss.2015.03.005`

[19] Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Logic **5**(3), 403–435 (Jul 2004). https://doi.org/10.1145/1013560.1013562, `http://doi.acm.org/10.1145/1013560.1013562`

[20] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: ACM Symposium on Principles of Programming Languages, POPL. ACM (1989)

[21] Prieur, C.: How to decide continuity of rational functions on infinite words. Theor. Comput. Sci. **276**(1-2), 445–447 (2002). https://doi.org/10.1016/S0304-3975(01)00307-3, `https://doi.org/10.1016/S0304-3975(01)00307-3`

[22] Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: CSL. Lecture Notes in Computer Science, vol. 4207, pp. 41–57. Springer (2006)