Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 276 (2011) 247–261

www.elsevier.com/locate/entcs

Ramified Corecurrence and Logspace

Ramyaa Ramyaa[a] and Daniel Leivant[a,b]

[a] *Indiana University*

[b] *LORIA Nancy*

Abstract

Ramified recurrence over free algebras has been used over the last two decades to provide machine-independent characterizations of major complexity classes. We consider here ramification for the dual setting, referring to coinductive data and corecurrence rather than inductive data and recurrence.
Whereas ramified recurrence is related basically to feasible time (PTime) complexity, we show here that ramified corecurrence is related fundamentally to feasible space. Indeed, the 2-tier ramified corecursive functions are precisely the functions over streams computable in logarithmic space. Here we define the complexity of computing over streams in terms of the output rather than the input, i.e. the complexity of computing the n-th entry of the output as a function of n. The class of stream functions computable in logspace seems to be of independent interest, both theoretical and practical.
We show that a stream function is definable by ramified corecurrence in two tiers iff it is computable by a transducer on streams that operates in space logarithmic in the position of the output symbol being computed. A consequence is that the two-tier ramified corecursive functions over *finite* streams are precisely the logspace functions, in the usual sense.

Keywords: Coinductive data, stream automata, corecurrence, lazy corecurrence, ramification, logarithmic space, implicit computational complexity

1 Introduction

1.1 Overview

Implicit computational complexity (ICC) deals with intrinsic properties of complexity classes, properties that do not refer directly to machine-based resources, such as computation time or space. That is, one matches complexity measures defined in terms of machine models resources, such as time and space, with declarative paradigms that are restricted along functionality, linearity, repetitions, flow control, or similar parameters. The benefits and potential applications of this research are well known (see e.g. [3,14]). Of particular practical interest is the characterization of computational complexity classes by restricted but natural declarative programming languages, since such languages guarantee complexity bounds automatically.

[1] {ramyaa,leivant}@indiana.edu

doi:10.1016/j.entcs.2011.09.025

A well known approach along these lines is *data ramification*, also known as *tiering*. One thinks of data as coming in varying computational strengths. For example, very large data may be thought of as a database to be queried, but too large to be used as a template to drive another recurrence; that is, a function's recurrence argument should be at a higher tier than its output. Data tiering has been used to characterize PTime [3,14], PSpace [15], as well as other feasible complexity classes.

Here we explore the ramification approach for coinductive, rather than inductive data, in particular streams rather than words. We identify a natural notion of logspace Turing machines over streams, and show that it computes those functions over streams that are definable by ramified corecurrence (using two tiers). More precisely, our data-type of choice is the set $S(\Sigma)$ of infinite as well as finite streams over a fixed finite alphabet Σ, i.e. the set generated coinductively (in a sense to be made precise below) from the nullary constructors ε and σ for each $\sigma \in \Sigma$, and the constructor *cons*, with the proviso that the first argument of *cons* is a symbol $\sigma \in \Sigma$. When ε is absent as constructor, we get the infinite streams only. As usual we write $\sigma : S$ for the stream $cons(\sigma, S)$. Evidently, our streams are merely a notational variant of words, infinite as well as finite, over the alphabet Σ.

The connections between coinduction and automata have been studied extensively in a category theoretic setting (see e.g. [21]). Corecurrence (without ramification) were similarly studied in [26,25].

Relations between corecursion and implicit computational complexity have also been explored. A program scheme based on safe recursion over streams was given in [5], but its computational complexity was not fully discussed. Feree et als. [9] use first order functional programs over streams to characterize complexity classes of functionals using second order polynomial interpretations. Semantic characterizations have been given for stream programs' termination and complexity in [10]. These characterizations are for streams over words or natural numbers rather than the digit streams we consider here. Finally, we have studied in [19] the relation between ramified corecurrence over words and the basic feasible functionals of Cook and Urquhart [8]. In the latter one mixes inductive and coinductive data, and ramification applies to both.

2 Machine transducers over streams

2.1 Finite transducers

We consider machine models for computing functions from streams to streams. Our basic machine model is the finite transducer (FT) over streams (often defined over ω-words instead), which allows for one-way reading of the input, and one-way writing of the output, with no requirement to either read or write during a computation step. [2] Note the differences between finite transducers, where acceptance conditions play no role, and familiar finite acceptors, such as Büchi automata. As for FTs over words, FTs over streams can differ in the number of cursors, in their allowed

[2] An equivalent formulation has the machine read a finite word and write a finite word at each step, where each of the two words may be empty [20].

movements, and in the output mechanism. We consider FTs that take possibly several inputs.

Examples. Here are some functions computable by simple FTs. We take the alphabet to be $\{0, 1, 2, 3\}$.

(i) Extract from the (single) input x the stream of symbols at even positions in x.

(ii) Extract from the input the sub-stream of symbols in $\{0, 1\}$. Note that the output, for an infinite input, may be finite, or even empty.

(iii) Merge two input streams.

(iv) For inputs x, y, output x until the first 3, and if one is found, proceed to output y.

(v) Sequential and subsequential transducers have been studied extensively (see e.g. [20] for an early survey). When the notion of an accepting state is removed, e.g. by considering all states as accepting, they are all FT in the present sense.

Formally, an r-ary *FT over Σ-streams* consists of a finite set Q of states, with a distinguished start state s, and a transition (partial) function $\delta : Q \times \Sigma^r \rightharpoonup Q \times \{1, \ldots, r\} \times \Sigma_\epsilon$ where $\Sigma_\epsilon := \Sigma \cup \{\varepsilon\}$. The intent is that $\delta(q, \boldsymbol{\sigma})$ consists of the next state, instructions as to which cursor is stepped forward, and an optional output symbol.

A *configuration* of a FT as above consists of a state and binary addresses for the r cursors. Thus each configuration is a finite syntactic object, albeit referring to input streams which may be infinite. The *initial configuration* is $\langle s, 0, \ldots, 0\rangle$. Write *Cfg* for the set of configurations. The transition partial-function δ determines, as usual, a partial-function $Yield : S^r \times Cfg \to Cfg \times \Sigma_\epsilon$ that maps the r input streams and a configuration to the next configuration and an optional output symbol. The *output* of a FT on input $\boldsymbol{s}$ is the stream of output symbols obtained from iterating Yield starting with the initial configuration.

Proposition 2.1 *If f and g are stream functions computed by FTs, then the composition $\lambda x.f(g(x))$ is also computed by a FT.*

The proof is straightforward, using auxiliary states that wait for output symbols of $g(x)$ to be obtained as needed to compute the output of $f(g(x))$.

We also consider a generalization of finite transducers to *jumping finite transducers (jumping FTs),* which may move an input cursor to the current position of another input cursor. Note that such jumps may include resetting cursors to the head of the input: we need only to maintain auxiliary cursors at the head of the inputs. Cook and Rackoff [7] introduced automata similar to reset automata, and a related model was introduced already in [22].

2.2 Turing transducers over streams

Turing transducers (TT) over streams extend FTs with an auxiliary read/write memory (work-tapes). Initially the work-tapes are empty, so the contents of each

work-tape is always a finite stream (equivalently, a word). The formal description of our Turing transducers and their operational semantics is a straightforward extension of the definitions above for FTs.

Turing transducers over streams can simulate Turing transducers for functionals (i.e. with functions as oracles), if their work-tapes are unrestricted. However, crucial differences surface when resources are considered, and when the real-time sequentiality of stream transducer contrasts with the random access underlying oracle Turing-machines.

We also consider *jumping Turing transducers (jumping-TTs)* which, like the jumping-FTs defined above, allow a reset of input cursors to the current position of other input cursors. Jumping Turing transducers are of importance to us here, because lazy corecurrence (see Definition 4 below) permits precisely the reinitialization of a computation to the current position of other cursors.

2.3 Space complexity of Turing transducers with and without jumps

We say that a jumping Turing transducer *operates in space* $f(n)$ if the computation, for the first n output symbols, does not involve work-tapes of length $> f(n)$. We say that a jumping Turing transducer is *log-space* if it operates in space $O(\log(n))$.

Examples.

(i) The Thue-Moore sequence [1] is generated by a logspace transducer without input (i.e. logspace generator).

(ii) Fix a polynomial $P(x)$ in one variable with integer coefficients. The stream function that on input x extracts the stream consisting of the entries of x in positions $p(n)$, $n = 1, 2, \ldots$, is computable in logspace.

(iii) In the previous example, let the output consist of the digits in positions $p(n)$ among the `0`/`1`entries of x.

(iv) $f(x)$ has in position n the first digit following a block of n consecutive `0`'s in the input.

(v) Consider the function that for input w yields the (possibly finite) stream whose n'th symbol τ is obtained by searching in w for the first symbol σ with n consecutive occurrences, and letting τ be the symbol following the first occurrence of σ in w. This function is computable in logarithmic space by a jumping-TT, but not by a TT without jumps.

(vi) Let g be the FT-computable function that returns its input with `1`'s omitted. Define $h(x) = g(f(x))$, where f is as in the previous example. That is, $h(x)$ is obtained by deleting `1`'s from $f(x)$. The n-th digit of $h(x)$ is the ℓ-th digit of $f(x)$, where ℓ might be exponentially (indeed, arbitrarily) larger than n. It follows that h is not logspace.

This example shows that the composition of a logspace-function with an FT-function may fail to be logspace. Thus the collection of logspace functions is not closed under composition.

In contrast to the last example above we have:

Proposition 2.2 *Let f, g be functions over streams. If f is logspace and g is computed by a jumping-FT, then $h(x) = f(g(x))$ is logspace.*

Proof. The proof is virtually the same as for Proposition 2.1. □

It is reasonable to consider a variant of Turing transducers where the output tape is two-way, with the cursor possibly revisiting an earlier output symbol. But for logspace transducers such an extension makes no difference: using additional work-tapes we can track the intended positions of cursors, and then recalculate the output symbols at such positions. The space used for each such calculation is immediately released at its end, so the entire process is still in log-space.

An alternative notion of logspace complexity for stream functions uses the parameter n above more explicitly, that is: the n-th entry of the output is computed from the inputs as well as the finite stream $\mathtt{1}^n$, using space $O(\log n)$. The extra argument $\mathtt{1}^n$ is not necessary, though, since an auxiliary work-tape can keep a count (in binary) of the outputs, and mimic operations on the input $\mathtt{1}^n$ using that auxiliary tape instead.

2.4 Simulating logspace with automata

We define variants of finite transducers that can simulate logspace Turing transducers, in much the same way as 2-way multi-cursor automata simulate logspace for words [11]. Here the extra input $\mathtt{1}^n$ stipulated above as an option plays a central role.

The idea for word automata is to code the contents of the work-tapes using auxiliary cursors on the input. That construction generalizes to logspace transducers over streams, provided we are given some finite stream (a "yardstick") on which the computation on the work-tapes can be simulated. In the special case where the input is finite, the input itself can serve as yardstick. In the general case, we can posit that when computing the n-th input entry the string $\mathtt{1}^n$ is given as extra input. As noted above, this is an innocuous assumption about logspace transducers. We say that such an automaton uses the input $\mathtt{1}^n$ as a *local counter.* Alternatively, we can posit as extra input the stream

$$\star =_{\mathrm{df}} \mathtt{1}0\mathtt{1}^2\mathtt{0}\mathtt{1}^3\mathtt{0}\cdots$$

that combines all local counters. We say that such an automaton uses a *global counter.* Finally, we might consider a computation model with an auxiliary output stream, on which the machine has 2-way read-only cursors.

The latter model seems to be fairly natural algorithmically. However, we focus here on functions defined by corecurrence, for which the use of output as an auxiliary computation space is alien. We shall therefore focus on machines with local counters.

An automaton with a local counter can use several cursors on the counter to simulate a work-tape of size $\log n$, including the position of the Turing transducer's cursors on it. When the Turing transducer is in space $\leq m \cdot \log n$, the simulation

of the work-tapes requires counting up to n^m, which can be done over $\mathtt{1}^n$ by using m cursors. An automaton with a global counter, whenever printing another output symbol, can peel 1's off the global counter, until a 0 is encountered, and use the next block of 1's as a yardstick.

We thus obtain,

Proposition 2.3 *If a stream function f is computed by a logspace (jumping-) Turing transducer, then it is computed by a (jumping, respectively) finite transducer using a 2-way local counter as well as by a (jumping, respectively) automaton using a 2-way global counter.*

We can further simplify the finite transducers used to simulate logspace Turing transducers, as follows.

Proposition 2.4 *Suppose a stream function f is computed by a jumping logspace Turing transducer. Then it is computed by a jumping transducer with a local counter, as well as by a jumping transducer with a global counter.*

Proof. By Proposition 2.3, it suffices to show that 2-way cursors on a finite counter can be simulated by cursors that can be reset, and that 2-way cursors on a global counter can be simulated by cursors that can be jumped to the position of other cursors.

Construct a jumping transducer M' that differs from M in that it simulates each backward move of a cursor c as follows. (a) Place an auxiliary cursor c' at the head-marker of c's tape, and step it forward; (b) step c and c' forward in tandem, until c reaches the end of the tape, at which time it is reset to the head of the tape; (c) step c and c' in tandem, until c' reaches the end of the tape. Thus c and c' both scan the entire finite work-tape once, with c lagging by one step, i.e. ending at a position preceding its initial one.

For a global counter, a jumping transducer M' simulates M as follows. After each output symbol the machine advances a reserved cursor to the next 0 of the counter input. The simulation then proceeds as for (1) above, with the next 0 used in place of ε to delimit the current $\mathtt{1}^n$ block from above, and the reserved cursor in place of the stream head to delimit that block from below. □

3 Ramified corecurrence

3.1 Corecurrence

Inductive data types, such as strings and lists, are defined as the least set closed under constructors. For example, the set $\mathbb{W} = \{0,1\}^*$ can be construed as the least set closed under the nullary constructor ε and the unary constructors 0 and 1. Correspondingly, (simultaneous) recurrence over a word algebra $\mathbb{W}$ allows for the definition of a vector $\boldsymbol{f}$ of functions from given function-vectors $\boldsymbol{g}_e$ and $\boldsymbol{g}_c$:

$$\begin{aligned} \boldsymbol{f}(\varepsilon, \boldsymbol{x}) &= \boldsymbol{g}_\epsilon(\boldsymbol{x}) \\ \boldsymbol{f}(\mathbf{c}(y), \boldsymbol{x}) &= \boldsymbol{g}_c(\boldsymbol{f}(y, \boldsymbol{x}), y, \boldsymbol{x}) \quad \mathbf{c} \text{ a (unary) constructor} \end{aligned} \tag{1}$$

The first argument of $\boldsymbol{f}$ drives the recurrence and is dubbed the *recurrence argument,* and the arguments of $\boldsymbol{g}_c$ corresponding to $(\boldsymbol{f}(y, \boldsymbol{x})$ are the *critical* arguments.

Coinductive data is obtained by a process dual to the generative definition of inductive data. In model-theoretic terms (rather than categorical terms) the *coinductive substructure* of a given structure $\mathcal{S}$ is the largest subset of the image of the constructors in $|\mathcal{S}|$, that is closed under destructors. Taking the same constructors as for $\mathbb{W}$, we obtain the set of finite and infinite words over the booleans. These are commonly recast as streams, with ε and *cons* as constructors, and where the first argument of *cons* is restricted to the alphabet Σ considered. We focus here on such streams, using *hd* and *tl* for the head- and tail-destructors.

Dual to the schema of recurrence, we have here the schema of *corecurrence,* defining function-vectors $\boldsymbol{f}$ over streams from given functions $\boldsymbol{g}$ and $\boldsymbol{h}$. The traditional, well-known, form of corecurrence is

$$hd(f_i(\boldsymbol{x})) = h_i(\boldsymbol{x}) \tag{2}$$
$$tl(f_i(\boldsymbol{x})) = f_j(\boldsymbol{g}(\boldsymbol{x}) \tag{3}$$

Thus, corecurrence uses the composition of the destructor and the defined function, rather than the composition of the defined function and the constructors, as in recurrence. The inputs to f given as outputs of $\boldsymbol{g}$ are the *critical arguments,* and the functions $\boldsymbol{g}$ are the *step-functions.*

The schema (2) is rather limited computationally, in that it requires that each computation cycle generates a new output entry. This excludes many of the functions computed even by finite transducers, as illustrated by the examples above. Drawing an analogy with recurrence, note that the step-functions $\boldsymbol{g}$ in (1) need not have a size-changing effect, e.g. $f(n+1)$ might equal $f(n)$ for all composite numbers n. This computational flexibility is restored in the following schema, which we dub *lazy corecurrence,* renaming the corecurrence schema above *strict corecurrence,* to clarify the distinction.

$$f_i(\boldsymbol{x}) = \mathit{if}\ hd(j_i(\boldsymbol{x}) = \varepsilon \tag{4}$$
$$\mathit{then}\ h_i(\boldsymbol{x}) : f(\boldsymbol{g}_i(\boldsymbol{x})) \tag{5}$$
$$\mathit{else}\ f(\boldsymbol{g}_i(\boldsymbol{x})) \tag{6}$$

That is, an output symbol is generated for the function f_i subject to a test j_i.

Thus, for example, over $\Sigma = \{0, 1, 2\}$ we can define

$$xtrct(x) = \mathit{if}\ hd(case(hd(x), 0, 0, \varepsilon, \varepsilon)) = \varepsilon \tag{7}$$
$$\mathit{then}\ hd(x) :\ xtrct(tl(x)) \tag{8}$$
$$\mathit{else}\ xtrct(tl(x)) \tag{9}$$

The function *xtrct* extracts from the stream input x the sub-stream consisting of `0`'s and `1`'s. Note that (2) would not do even for functions computed by finite transducers, as in the examples above,

Lazy corecurrence is non-productive, in the sense that it may yield from an stream as input an output that fails to be infinite. This is a form of type failure. We have, however,

Lemma 3.1 *A function f defined by lazy corecurrence is the composition of a function f' defined by strict corecurrence with a function e defined by a FT.*

Proof. Let f' output a blank symbol whenever f generates no output symbol, and define e to erase such blanks from its input. □

The *strict corecursive functions* over streams are generated from constants, the destructors and projections functions, and the schemas of composition and strict corecurrence. Note that strict corecursive functions are always productive, and thus do not include the non-productive examples above of FT-computable functions. On the other hand, there are computable productive functions that are not strict corecursive; for example, a construction dual to the definition of Ackermann function yields such functions.

3.2 Ramification

Safe recurrence was initiated by Bellantoni and Cook [3]. It was slightly generalized in [14], where the two tiers of safe recurrence (safe and normal) are just the first two in a series of tiers. That is, in ramified recurrence one posits copies ("tiers") $\mathbb{W}_0, \ldots, \mathbb{W}_i, \ldots$ of the algebra $\mathbb{W}$ of finite words, with the intent that each copy consists of strings that are computationally "stronger" than the ones in lower tiers. *Ramified recurrence* is the schema of recurrence (1) above, with the proviso that the recurrence argument is at a tier higher than that of the critical arguments. Thus, ramified recurrence prevents the use of critical arguments as recursive arguments.

The *ramified recursive functions* over $\mathbb{W}$ are generated from initial functions by ramified (i.e. tier-respecting) composition and ramified recurrence. The initial functions are constructors, projections, and the definition-by-cases function, which for $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ reads

$$case(w, x, y, z) = \begin{cases} x \text{ if } w = \varepsilon \\ y \text{ if } w = \mathtt{0}u \\ z \text{ if } w = \mathtt{1}u \end{cases}$$

Each of these is considered polymorphic, i.e. for each i mapping arguments of tier i to a result of tier i. One exception is the *case* function, for which the first argument can be of any tier.

The ramified strict-corecursive (respectively, lazy-corecursive) functions are generated from initial functions using tier-respecting composition and ramified strict-corecurrence (respectively, lazy-corecurrence). The initial functions here are the constants finite streams, the destructors *hd* and *tl* (tier respecting), projections and the branching function

$$case(s, x, y, z) = \begin{cases} x \text{ if } s = \varepsilon \\ y \text{ if } hd(s) = 0 \\ z \text{ if } hd(s) = 1 \end{cases}$$

We stipulate that for $\sigma \in \Sigma_\epsilon$ $hd(\sigma) = \sigma$ and $tl(\sigma) = \varepsilon$. Again, each of these functions is considered polymorphically, that is for each t it maps arguments of tier t to a result of tier t. In particular, the constants are polymorphic for all tiers. Again, an exception is the first argument of *case*, which can be taken at any tier.

We focus here on ramification that uses two tiers only. This implies that the step functions used in corecurrence must be *flat,* that is tier-preserving. In fact:

Lemma 3.2 *A flat ramified function definition can be converted, by re-assignment of tiers only, into a flat definition with the output and all inputs of tier 0.*

Example 3.3 Consider the simultaneous strict-corecurrence

$$\begin{aligned} hd(r(x)) &= hd(x) & \qquad hd(r'(x)) &= hd(x) \\ tl(r(x)) &= r'(x) & \qquad tl(r'(x) &= r(tl(x)) \end{aligned}$$

That is, the function r outputs the input stream with every entry repeated. The definition above is ramified, by letting the outputs' tier be greater than the input tier.

Now define

$$\begin{aligned} hd(e(x,y)) &= y \\ tl(e(x,y)) &= e(tl(x), r(y)) \end{aligned}$$

The function e is thus strict-corecursive, but its definition can not be ramified: e takes as second input both y and $r(y)$, and since r is tier-increasing, the second input's tier cannot be defined.

We show in Proposition 4.5 below that every 2-tier ramified lazy-corecursive function between finite streams is logspace in the usual sense, i.e. in the size of the input. But e maps $(1^n, 1^n)$ to a stream of length 2^n, so e is not ramified lazy-corecursive, let alone ramified strict-corecursive.

We argued above that the simulation of a work-tape by an automaton must, unless the input is finite, rely on a counter supplied as input. The same applies to corecurrence. We thus say that a function $f : S^r \to S$ is defined by a *counter (strict/lazy) corecurrence* if there is a function f' of $r+1$ arguments, defined by (strict/lazy) corecursive function, and such that n-th entry of $f(\boldsymbol{x})$ is $f'(\boldsymbol{x}, 1^n)$.

3.3 Ramification from a foundational viewpoint

The duality between inductive and coinductive data is brought out in second order logic *SOL*, and so ramification of both recurrence and corecurrence can be better understood through a ramified variant *RSOL* of *SOL*, a formalism studied by logicians for over 50 years [24]. The impredicative nature of *SOL* is well known: a set-quantified formula $\forall X\ \varphi[X]$ can be instantiated to any second-order definable set X_0, even if X_0 is defined in term of the formula $\forall X\ \varphi$ itself. To avoid this, *RSOL* assigns tiers to set and relational variables, and a variable X of a given tier t cannot be instantiated to a relation X_0 whose definition uses quantifiers of tier $> t$.

The second order definition of the set $\mathbb{W} = \{0,1\}^*$ is second-order universal:

$$W[x] \equiv_{\text{df}} \forall Q.\, \text{Cl}[Q] \to Q(x)$$

where

$$\text{Cl}[Q] \equiv_{\text{df}} Q(\varepsilon) \wedge \forall y.\, Q(y) \to Q(0y) \wedge Q(1y)$$

In *RSOL* this definition has a separated variant for each tier t, W_t, in which Q is assigned the tier t. This gives rise to a sequence of definitions of $\mathbb{W}$ at each tier t. with Q above taken to be of tier t.

In proving that a recursive definition yields a well-typed function $f : \mathbb{W} \to \mathbb{W}$, one proves the second order formula $W[x] \to W[f(x)]$. If that proof can be ramified, then the quantifier $\forall Q$ in the premise $W[x]$ is instantiated must be higher than that of the conclusion $W[f(x)]$, i.e. one proves $W^t[x] \to W^q[f(x)]$ for some $t \geq q$.

Thus, if f is defined by recurrence and is proved correct in *RSOL* as above, then the recurrence argument is at a tier higher than the tier of the critical arguments. This gives rise to the ramified variation of the recurrence schema.

A dual analysis applies to coinductive data. The set of finite and infinite streams is defined by the second order formula

$$S[x] \equiv \exists R.\, Pl[R] \wedge R(x)$$

where

$$Pl[R] \equiv \quad \forall z.\, R(z) \to z = \varepsilon \ \vee\ \exists y\, (R(y) \wedge (z = 0 : y \vee z = 1 : y)) \qquad (10)$$

In *RSOL* the formula S has a variant S_t at each tier t, corresponding to the tier assigned to the variable R. The second order statement of the productivity of a function f, is $S[x] \to S[f(x)]$. In *RSOL* we have the ramified variants. $S^t[x] \to S^q[f(x)]$. The quantifier $\exists R^t$ in the premise is used to construct a definition of a relation Q_0 of tier t to yield $\exists R^q$ in the conclusion. It follows that $q \geq t$. For the schema of corecurrence, this means that the critical arguments, i.e. the inputs, must be of tier lower than the tier of the output. Thus, ramified corecurrence requires a tier increase from input to output, just the opposite of ramified recurrence.

4 Logspace stream computations and ramified corecurrence

4.1 Logspace stream functions are 2-tier ramified corecursive

Proposition 4.1 *Every logspace function $f : S^r \to S$ is definable by 2-tier lazy corecurrence (with a local or global counter).*

Proof. By Proposition 2.4 it suffices to show that if f is computed by a jumping finite transducer M with a local counter, then it is 2-ramified corecursive.

Let M be a jumping finite transducer computing f, with r inputs and k cursors. For each state q of M fix a different finite stream $\bar{q}$. We consider an $(1 + r + k)$-ary function f, intended to return for inputs $\bar{q}, x_1, \ldots, x_r, z_1, \ldots z_k$ the stream generated by M when computation for input $\boldsymbol{x}$ is launched at state q with inputs $\boldsymbol{x}$, and with the cursors at the heads of $z_1, \ldots, z_k$.

The function f is defined by corecurrence as follows. Let $h(\bar{q}, \boldsymbol{z})$ be the output symbol of M when in state q with the cursors at symbols $\boldsymbol{\sigma} = hd(\boldsymbol{z})$, if such an output is produced; and $= \varepsilon$ otherwise. Similarly, let $j(\bar{q}, \boldsymbol{z})$ be ε if M produces an output as above, and $= 0$ otherwise. Also, for each cursor c let $g_c(\bar{q}, \boldsymbol{x}, \boldsymbol{z})$ be $tl(z_c)$ if M with $q, \boldsymbol{\sigma}$ as above steps the c-th cursor forward, $= z_i$ if M resets the c-th cursor to the head of input z_i, and $= z_c$ otherwise. Finally, let $p(\bar{q}, \boldsymbol{z})$ be the updated state of M for a transition as above.

Note that these functions area all explicitly definable using branching and destructors. They are therefore definable as ramified functions from tier 0 to tier 0. (We indicate here the inputs $\bar{q}$ to convey the intention more clearly, but the formal definition, for a variable q, is likewise obtained by branching and destructors.)

We now have the lazy corecursive definition

$$\hat{f}(q, \boldsymbol{x}, \boldsymbol{z}) = \mathit{if}\ j(q, \boldsymbol{z}) = \varepsilon \tag{11}$$

$$\mathit{then}\ h(q, \boldsymbol{z}) : \hat{f}(p(q, \boldsymbol{z}), \boldsymbol{x}, \boldsymbol{g}(q, \boldsymbol{x}, \boldsymbol{z})) \tag{12}$$

$$\mathit{else}\hat{f}(p(q, \boldsymbol{z}), \boldsymbol{x}, \boldsymbol{g}(q, \boldsymbol{x}, \boldsymbol{z})) \tag{13}$$

$$\tag{14}$$

Since all functions used here are ramified from tier 0 to tier 0, it follows that the corecurrence (11) is ramified, with 1 as output tier.

Now let $z_1, \ldots, x_r$ be the r input values, and $z_1, \ldots, z_k$ be those x_is (possible repeated) that correspond to the initial positions of the cursors on the inputs. Since $f(\boldsymbol{x}) = \hat{f}(\bar{s}, \boldsymbol{x}, \boldsymbol{z})$, where s is the initial state of M, this proves the Proposition. □

4.2 Every 2-ramified counter-corecursive function is logspace

We proceed to prove the converse of Proposition 4.1.

Suppose $f : S^p \times S^q \to S$. We say that a function f is *narrow in its first p arguments* if there is some $m \geq 0$ such that for all k, $ent_k f(s_1, \ldots, s_p, \ldots)$ is independent from entries $s_1 \ldots s_p$ at positions $> m$. We say that f is *logspace in its last q arguments* if f, as a function of these arguments, is computable in logspace.

Proposition 4.2 *Suppose $f : S_1^p \times S_0^q \to S_i$ is a 2-ramified lazy-corecursive function. Then f is narrow in its arguments of tier i, logspace in arguments of tier $< i$, and independent of arguments of tier $> i$.*

Proof. Induction on the definition of f as a 2-ramified function. The destructors and *case* function preserve tiers, and they are indeed narrow in their arguments. The reasoning for constants and projection is straightforward.

Consider tiered composition: $f(\boldsymbol{x}) = g(h(\boldsymbol{x}), \boldsymbol{x})$. If the output tier of f, i.e. of g, is 0, then, by IH, g is narrow in its arguments of tier 0, and independent of its arguments of tier 1. If the output tier of h is also 0, then h is narrow in its arguments of tier 0, and independent of its arguments of tier 1. It follows that the same holds of f. If, on the other hand, the output tier of h is 1, then g is independent of h, and so f satisfies the Proposition's statement outright.

Finally, suppose that f is defined by ramified lazy corecurrence, as in 4. Then, by the tiering conditions, the output tier of f must be 1, and the output tiers of the

functions $\boldsymbol{g}$ must all be 0. So, by IH, the functions $\boldsymbol{g}$ are narrow in their inputs. Each such g can be represented directly by a cursor, since assignments and composition of tails are all fixed instructions to cursors, which can be coded in the transition table. But then $ent_k(f(\boldsymbol{x}))$, i.e. $hd(\boldsymbol{g}^n(\boldsymbol{x}))$, can be computed by a logspace Turing transducer that keeps a binary count i while calculating successively the first $i \cdot m$ entries of $\boldsymbol{g}^i(\boldsymbol{x})$. □

Combining Propositions 4.1 and 4.2, we get:

Theorem 4.3 *A function over streams is 2-ramified lazy-corecursive (with counters) iff it is logspace.*

Corollary 4.4 *A function f over streams is logspace iff it is the composition of a 2-ramified strict-corecursive function with a FT-computable function.*

Proof. If f is logspace then it is 2-ramified lazy-corecursive, by Theorem 4.3. In fact, the proof shows that f is definable by a single ramified lazy-corecurrence, from a function which is narrow in its arguments and therefore defined without corecurrence altogether. It follows from Lemma 3.1 that f is the composition of a strict corecursive function with an FT-computable function.

For the converse, suppose that $f(\boldsymbol{x}) = g(f'(\boldsymbol{x}))$ where f' is 2-ramified strict-corecursive and g is FT-computable, By Theorem 4.3 f' is logspace, and so by Proposition 2.1 f is logspace is well. □

4.3 Functions over finite streams

A natural question is to relate output-based complexity for finite streams with input-based complexity for functions over words. When the input itself is finite, it makes little sense to posit an extra input $\mathtt{1}^n$. We therefore dispense in this context of the extra input $\mathtt{1}^n$, and use the finite input instead.

Since the size c of configurations is logarithmic in the size n of the input, it follows that the number of distinct configurations is polynomial in n. It follows that the output must be rational with a period which is polynomial in n. In particular, if the output is finite, then it must be of size polynomial in n. Thus $\log(k)$ for $k =$ the length of the output is itself $O(\log(n)$. We have thus proved:

Proposition 4.5 *Let f is a 2-tier ramified primitive corecursive function that for each finite input has a finite output. Then f is logspace in the size of the input.*

Theorem 4.6 *A function over finite streams is 2-ramified lazy-corecursive iff it is logspace in the size of the input.*

Proof. The forward direction is given by Proposition 4.5. For the converse, assume that a function f over finite streams is logspace in the size of the input. Then the output is of size polynomial in the size of the input. So the computation of the n-th output entry is logspace in the size of the input, which implies that the output is computable in logspace in the size of the input. □

There are several known implicit characterizations of the *languages* decidable in logarithmic space. Jones [12] presents a characterization in terms of imperative

while-programs over lists, without recursion nor use of the *cons* function. Kristiansen [13] gives a characterization based on a simple recursion scheme that does not use *cons*. Bonfante [4] uses tail recursion (which bears structural similarity to corecursion), using tiering to limit branching.

Several implicit characterizations of logspace use the fact that the size of a binary string is the logarithm of its binary-value, and adapt the characterizations of PTime by ramification to logspace [18,16,6]. Bellantoni [2] also uses a variant of recurrence, where function output is required to be in unary, whereas the inputs are in binary. Neergaard [17] uses linearity constraints applied to general safe-recursion, as opposed to safe-recurrence (i.e., safe primitive-recursion). A restricted version of linear logic is used to characterize logspace in [23].

The main novelty of our characterization is in establishing a link between logspace and corecurrence, rather than special forms of recurrence. This new link is of particular interest in the modern context of computing devices sharing vast databases (such as the entire internet), for which infinite data-objects such as streams might be viewed as an appropriate mathematical abstraction.

5 Conclusion

Natural models for computing over streams should bring out the sequential, on-line, consumption of the input and generation of the output, in contrast to the random-access nature of computing with function-oracles. It is sometimes argued that the only relevant computation model that follow the on-line paradigm are finite automata, a position that agrees with a straightforward category-theoretic view of corecursion. However, from a computational viewpoint it is quite natural to enhance this paradigm with the ability to recognize the computation locale, i.e. using addresses for the position of the output symbol being calculated. This seems to underlie already several natural stream constructions, such as the Thue-Moore sequence, as pointed above. From a practical viewpoint equipping finite transducers with such a rudimentary counting' ability is completely harmless, since $\log n$ is a small number even when n is the number of particles in the universe. Logspace Turing transducers between streams seem therefore no less natural than finite transducers as mathematical abstraction of actual computing over streams.

On the other hand, we considered here declarative programs for stream computation, in the guise ramified corecurrence. Whereas 2-tier ramified recurrence over words yields polynomial time [3], our main technical result establishes a dual theorem for computation over streams: the functions definable using 2-tier corecurrence over streams are precisely those that are computable in space logarithmic in the position of the output symbol computed. An extra ingredient has to be included in the latter correspondence, because the auxiliary memory of a Turing transducer is a finite word, not a stream, and finite words cannot be generated in a corecursive setting. We addressed this issue by allowing corecurrence to also refer to one finite "counter" of size n, when computing the n-th output symbol.

We have related logspace Turing transducers to ramified corecurrence using only

two tiers. The situation is far more complex when ramified recurrence may use arbitrary and different tiers for the inputs. We plan to prove elsewhere that the functions obtained then are precisely the functions definable by composition from logspace stream-functions.

We also argued that ramification of both recurrence and corecurrence can be construed in the context of ramified second order logic (or, equivalently, of λ-definability in the ramified version of system $\mathbf{F}_2$). We intend to develop elsewhere a combined setting for ramifying simultaneously recurrence and corecurrence; this should hopefully dispense us from using artifacts such as counters.

In summary, we feel that the present work, while establishing a duality between polytime for recurrence and logspace for corecurrence, leads to interesting questions about computation models for streams, the notion of ramification for declarative programs, and the relation between the two.

References

[1] J.P. Allouche and J. Shallit. *Automatic sequences: theory, applications, generalizations*, pages 152–153. Cambridge University Press, 2003.

[2] Stephen Bellantoni. *Predicative recursion and computational complexity.* PhD thesis, University of Toronto, 1992.

[3] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[4] Guillaume Bonfante. Some programming languages for logspace and ptime. In *AMAST*, pages 66–80, 2006.

[5] Michael J. Burrell, Robin Cockett, and Brian F. Redmond. Pola: a language for ptime programming. In *Logic and Computational Complexity, Workshop Proceedings*, 2009.

[6] Peter Clote and Gaisi Takeuti. Bounded arithmetic for nc, alogtime, l and nl. *Ann. Pure Appl. Logic*, 56(1-3):73–117, 1992.

[7] Stephen A. Cook and Charles Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM J. Comput.*, 9(3):636–652, 1980.

[8] Stephen A. Cook and Alasdair Urquhart. Functional interpretations of feasible constructive arithemtic. *Annals of Pure and Applied Logic*, 63:103–200, 1993.

[9] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Interpretation of stream programs: Characterizing type 2 polynomial time complexity. In *ISAAC*, pages 291–303, 2010.

[10] Marco Gaboardi and Romain Péchoux. Global and local space properties of stream programs. In *Proceedings of the First international conference on Foundational and practical aspects of resource analysis*, FOPARA'09, pages 51–66, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Inf.*, 1:336–344, 1972.

[12] Neil D. Jones. Logspace and ptime characterized by programming languages. *Theor. Comput. Sci.*, 228:151–174, October 1999.

[13] Lars Kristiansen. Neat function algebraic characterizations of logspace and linspace. *Computational Complexity*, 14:72–88, 2005.

[14] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, Perspectives in Computer Science, pages 320–343. Birkhauser-Boston, New York, 1994.

[15] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity ii: Substitution and poly-space. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500. Springer, Berlin and Heidelberg, 1995.

[16] John Lind and Albert R. Meyer. A characterization of log-space computable functions. *SIGACT News*, 5:26–29, 1973.

[17] Peter Møller Neergaard. A functional language for logarithmic space. In *APLAS*, pages 311–326, 2004.

[18] Isabel Oitavem. Logspace without bounds. In Ralf Schindler, editor, *Ways of Proof Theory*, Ontos Series in Matematical Logic, chapter Logspace without bounds, pages 355–362. Ontos Verlag, 2010.

[19] Ramyaa Ramyaa and Daniel Leivant. Feasible functions over co-inductive data. In *Workshop on Logic, Language, Information and Computation*, pages 191–203, 2010.

[20] Christophe Reutenauer. Subsequential functions: Characterizations, minimization, examples. In Jürgen Dassow and Jozef Kelemen, editors, *IMYCS*, volume 464 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 1990.

[21] Jan J.M.M. Rutten. Automata and coinduction (an exercise in coalgebra). In Davide Sangiorgi and Robert de Simone, editors, *Ninth International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.

[22] Walter J. Savitch and Paul M.B. Vitányi. Linear time simulation of multihead turing machines with head-to-head jumps. In *ICALP*, pages 453–464, 1977.

[23] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS*, pages 411–420, 2007.

[24] Kurt Schütte. *Proof Theory*. Springer-Verlag, Berlin, 1977.

[25] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica, Lith. Acad. Sci.*, 10:5–26, 1999.

[26] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). In *Ninth Nordic Workshop on Programming Theory*, 1998.