The computational contents of ramified corecurrence

Daniel Leivant[1] and Ramyaa Ramyaa[2]

[1] Indiana University Bloomington, leivant@indiana.edu
[2] Wesleyan University, ramyaa@wesleyan.edu

Abstract. The vast power of iterated recurrence is tamed by data ramification: if a function over words is definable by ramified recurrence and composition, then it is feasible, i.e. computable in polynomial time, i.e. any computation using the first n input symbols can have at most $p(n)$ distinct configurations, for some polynomial p. Here we prove a dual result for coinductive data: if a function over streams is definable by ramified *corecurrence,* then any computation to obtain the first n symbols of the output can have at most $p(n)$ distinct configurations, for some polynomial p. The latter computation is by multi-cursor finite state transducer on streams.
A consequence is that a function over *finite* streams is definable by ramified corecurrence *iff* it is Turing-computable in logarithmic space. Such corecursive definitions over finite streams are of practical interest, because large finite data is normally used as a knowledge base to be consumed, rather than as recurrence template. Thus, we relate a syntactically restricted computation model, amenable to static analysis, to a major complexity class for streaming algorithms.

1 Introduction

Implicit computational complexity relates resource-based complexity classes of functions and languages to declarative paradigms, restricted along various conceptual parameters, such as functionality, linearity, repetition, and flow control. The theoretical and practical benefits of this research abound, notably in leading to static analysis of the computational complexity of declarative programs.

A well known approach along these lines is *data ramification*, also known as *tiering*. Here one construes data as coming in varying computational strengths. For instance, querying a large database might be feasible, but using it to drive a recurrence would not. This is reflected in a requirement that a function's recurrence argument should be computationally stronger than its output, i.e. at a higher tier. This approach was used to characterize major complexity classes such as PTime [1, 9], and PSpace [13].

In [14] we initiated an exploration of ramified declarative programming over coinductive data, such as streams, rather than inductive data, such as words. We showed that functions defined by ramified corecurrence and composition using just *two tiers* are feasible, in the sense of being computable by finite state

transducers (with cursor jump). Moreover, such transducers can be simulated by Turing-transducers over streams that operate in logarithmic space with respect to the *output,* that is: to compute the n-th entry of the output requires auxiliary computation space of size $O(\log n)$.

The analysis of corecurrence ramified into an arbitrary number of tiers requires a more foundational approach. We introduce here notions of locality, weak-locality and continuity for machines over streams, and show that a function defined by ramified corecurrence and composition, with output tier t, is computable by a finite-state transducer (with jumps) which is continuous (with polynomial moduli of continuity) in arguments of tiers $< t$, weakly-local in arguments of tier t, and local in arguments of tiers $> t$. These properties are used to show that if a function f is defined by corecurrence from step-functions that are so defined, then f is computable by a continuous finite-state transducer (with jump).

Our results reveal a striking duality between ramified recurrence and ramified corecurrence. If a function f over words is definable by ramified recurrence then it is polynomial time or — equivalently — any computation of f that uses the first n input symbols has at most $p(n)$ distinct configurations (p a polynomial). Dually, if a function over streams is definable by ramified *corecurrence,* then any computation to obtain the first n symbols of the output can have at most $p(n)$ distinct configurations (p a polynomial). There is, however, an asymmetry: read-only inputs are given, whereas write-only outputs are not. corecurrence the configurations must, in addition, be of size logarithmic in the n, because the input is given, $p(n)$ entries input entries that might occur in configurations are determined merely by their address. This contrasts with recurrence: the output is not given, and so the $p(n)$ output entries that might occur in configurations may well be different.

These results have further consequences for functions over *finite* streams: such a function is definable by ramified corecurrence (in any number of tiers) iff it is Turing-computable (as a function over words) in logarithmic space, in the usual sense. Referring to finite streams may seem at first blush to be an oxymoron. Indeed, finite data is commonly identified with textual data, or more generally data generated inductively from constant values by iterating finite closure rules. A salient property of inductive data of that sort is their use to drive the recurrence schema associated with the corresponding generative process; for example, the recurrence (i.e. "primitive recursive") schema for the natural numbers. However, the increasing relevance of finite, but very large data, suggests an alternative viewpoint of finite data, that emphasizes access to data-elements. Viewed from that angle, it becomes relevant and interesting to consider finite instances of coinductive data. In particular, *finite boolean streams,* are extensionally similar to words, but their computational behavior is coinductive: while a computation over words gets as input complete words and produces complete words as output, a computation over streams produces its output piecemeal, using pieces of its input(s). Indeed, important applications that involve computing over very large data are modeled better by streams than by words (see for example [15, 6]). Thus, ramified corecurrence lends credence to the importance

and stability of log-space computing over large data, which is what all of us do daily in our use of the internet.

In summary, our results elucidate the foundational inter-relations between coinductive data, ramification, finite-state stream-transducers, and log-space computing, while providing a static-analysis method for establishing the feasibility of functional programs over streams.

2 Finite transducers on streams

2.1 Jumping finite transducers

Fix a finite alphabet $\Sigma = \{a_0, a_1, \ldots, a_\ell\}$ ($\ell \geq 2$). The set of *streams over* Σ, denoted $S(\Sigma)$ or simply S when Σ is clear, is defined coinductively by the closure condition:

$$\sigma \in S \implies (\exists a \in \Sigma)\, (\exists \tau \in S)\, \sigma = a : \tau$$

The basic machine model for computing functions from streams to streams is the *finite stream transducer (FT)* over an alphabet Σ. A FT reads its input stream one-way at multiple cursors ("heads") and writes its output stream one-way forward. The read is optional (i.e. read ε is possible), and so is the write. We also consider a less restrictive variant of FTs, the *jumping finite transducer (JFT)*. Here a cursor that scans the input may be re-positioned ("jump") to the current position of another cursor. Neither FTs nor JFTs can detect coincidence between two cursors. Using such detection, JFTs could simulate two-way cursors on the input, and consequently be Turing complete (on infinite streams).

Formally, an *r-ary JFT over Σ-streams* F consists of a finite set Q of states, a distinguished *start state* $s \in Q$, a finite set $C = \{c_1, \ldots, c_k\}$ of *cursors,* an initial *cursor configuration* $\gamma : C \to \{1, .., r\}$, a *transition* partial-function, and an *output* partial function. The transition partial-function

$$\delta : \; Q \times (C \to \Sigma) \;\; \rightharpoonup \;\; Q \times (C \to M)$$

refers to the set of *moves* $M = C \cup \{+\}$. When $\delta(q, \kappa) = \langle p, \alpha \rangle$ we also write $q \xrightarrow{\kappa(\alpha)} p$. The intent is: an argument $\kappa : C \to \Sigma$ gives the source values of the cursors; an action $\alpha : C \to M$ instructs each cursor c to jump to the position of cursor $\alpha(c)$, if $\alpha(c) \in C$, or to step forward if $\alpha(c) = +$.

The output partial-function

$$\mathcal{O} : \; Q \times (C \to \Sigma) \;\; \rightharpoonup \;\; \Sigma$$

indicates the output symbol, if any, that F emits.

To give the formal semantics of JFTs we refer to *configurations,* each consisting of a state and a mapping $\pi : C \to [1..r] \times \mathbb{N}$, that assigns to each cursor c a stream index $i \in [1..r]$ and an address (in binary) on that stream. Note that configurations are finite objects, and do not refer to input or output stream as a whole.

Let *Cfg* be the set of configurations. The *initial configuration* is $\beta_0 = \langle s, \pi_0 \rangle$, where $\pi_0(c) = \langle \gamma(c), 0 \rangle$.

Given infinite streams $\sigma_1, \ldots, \sigma_r$ as inputs, δ determines a partial *Yield* function

$$\mathit{Yld}_{F,\tilde{\sigma}} : \ \mathit{Cfg} \rightharpoonup \mathit{Cfg}$$

that maps a configuration to its successor configuration, as described informally above.[3] When $\beta' = \mathit{Yld}_{F,\tilde{\sigma}}(\beta)$ we also write $\beta \Longrightarrow_{F,\tilde{\sigma}} \beta'$. Thus, each configuration β generates a (finite or infinite) stream of configurations,

$$T(\beta) = \beta_1 : \beta_2 : \cdots$$

dubbed the *trace for* β, where β_1 is β and $\beta_i \Longrightarrow_{F,\tilde{\sigma}} \beta_{i+1}$. The trace $T(\beta)$ is finite when δ is undefined for its last configuration.

The output function $\mathcal{O}$ determines, for input streams $\tilde{\sigma}$, a *partial* function

$$\mathit{Outbit}_{F,\tilde{\sigma}} : \ \mathit{Cfg} \rightharpoonup \Sigma$$

that maps some configuration to symbols $a \in \Sigma$. The *output stream* of F for inputs $\tilde{\sigma}$ is obtained from collecting into a stream the output symbols emitted by $\mathit{Outbit}_{F,\tilde{\sigma}}$ for the trace of F for $\tilde{\sigma}$.

2.2 Composition of JFTs

We consider functions over the two base type Σ (the alphabet symbols) and S (streams over Σ). Consider the schema of typed composition of functions. $f(\tilde{x}) = d(e_1(\tilde{x}), \ldots, e_k(\tilde{x}))$. Using auxiliary variables, this can be reduced to the schema $f(\tilde{x}) = d(e(\tilde{x}), \tilde{x})$.

Theorem 1. *If d, e are JFT-computable functions over Σ and S, then so is $f(x_1, \ldots x_r) = d(e(\tilde{x}), \tilde{x})$.*

Proof. Suppose $d : S \times \tau_1 \times \cdots \times \tau_r \to S$ is computed by a JFT D, and $e : \tau_1 \cdots \times \tau_r \to S$ is computed by a JFT E, where the τ_i's are S or Σ.

We construct a JFT F to compute f, using a copy of D, as well as a copy E_c of E for each cursor c that D maintains on its first (i.e. composition-) argument, which we take to be x_0. Each such copy is intended to represent E producing the n-th entry of $e(\tilde{x})$, where n is the current position of the cursor c. Accordingly, F also maintains internally the value $\sigma_c \in \Sigma$ at that position.

F starts by initializing each σ_c to the first output symbol of E. It then moves on to simulate D. Where D would step forward a cursor c on D's first argument F runs E_c until the next output symbol σ is produced, leaves the internal configuration of E_c as reached, and updates σ_c to σ. Where D would read the value at c of its i-th input, F supplies σ_c as that value. Where D would relocate cursor c to the position of cursor c', F sets the configuration of E_c (internal states, cursor positions, and σ_c) to the configuration of $E_{c'}$. □

[3] We use tilde for vectors.

2.3 Locality and continuity properties of JFTs

Equational computing systems, such as the primitive corecursive functions, permit effortless copying of an input to the output, an operation which a JFT must carry out by an infinite entry-by-entry transfer. To better reflect the ease of equational copying we consider a variant of JFTs with an added operation $\mathbf{Go}_c$, whose semantics is: *The output from this point on is the stream starting with the current position of cursor c.* No computation power is added, of course, since each invocation of $\mathbf{Go}_c$ can be replaced by a trivial loop.

We say that a JFT F is *local* on an argument (i.e. input) x if there is a bound b (uniform with respect to all remaining inputs) such that F does not move cursors beyond x's b-th entry, and does not invoke a $\mathbf{Go}_c$ operation for any cursor c residing on x. F is *weakly-local* on x if it is local as above, except that F's output is obtained by a **Go** for some cursor on x (necessarily abiding by the locality condition, i.e. at a position $\leq b$). Note that this violates locality: sufficiently far-out entries of the output will depend on far-out entries of x, but in a very simple way: by identity. In either case, we refer to b as the *bound* of F on x.

We say that F is *continuous on x, with modulus $\omega : \mathbb{N} \to \mathbb{N}$,* if for each $n \geq 1$, while calculating its n-th output symbol F does not move cursors beyond the $\omega(n)$-th entry of x. We say that F *has degree k on argument y* if it has a modulus ω of order $O(n^k)$. F is *polynomial* on argument y if it has degree k on y for some k.

The next Lemma shows how the locality and continuity properties of functions determine those properties for their composition.

Lemma 1. *Assume the premises of Theorem 1, and let F be the JFT defined in its proof. Let y be one of the variables $x_1, \ldots, x_r$.*

1. *If D is local on x_0, or E is local on y, then F has the same property on y as D on y: local, weakly-local, or continuous with modulus ω.*
2. *If D has modulus ω_0 on x_0 and ω_1 on y, whereas E has modulus ω_E on y, then F has modulus $\max(\omega_0 \circ \omega_E, \omega_1)$ on y.*
   In particular, if ω_E and ω_0 are both constants, i.e. D is weakly-local on x_0 and E is weakly-local on y, and if D has modulus ω on y, then F has a modulus of order $O(\omega)$ on y.

Proof.

1. If D is local on z then F needs to access y via E only for a fixed finite set of entries of E's output, and since E is continuous, that means a fixed number of positions of y. On the other hand, if E is local on y, then F access y via E only for a fixed finite set of entries, regardless of D's queries for the output of E. In either case, the use of y by F is dominated by the direct access of D to y.
2. To calculate its n-th output symbol F reads $\omega_y(n)$ symbols of y when accessing y directly. It also needs to identify the first $\omega_z(n)$ symbols of the output of E, which calls for reading $\omega_E(\omega_z(n))$ entries of y.

□

3 Corecurrence

3.1 Stream functions defined by corecurrence

We continue to refer to an alphabet $\Sigma = \{a_0, a_1, \ldots, a_\ell\}$ and to the set S of streams over Σ. The scheme of corecurrence over streams provides a definition of a function $f : S^r \to S$ from given functions $h : S^r \to \Sigma$ and $g_i : S^r \to S^r$ $(i = 1..r)$:

$$f(\tilde{x}) = h(\tilde{x}) : f(\tilde{g}(\tilde{x})) \tag{1}$$

More generally, a function-vector $\tilde{f} = (f_1, \ldots, f_m)$ is defined from functions $\tilde{h}$, $\tilde{g}_1$, …, $\tilde{g}_m$:

$$f_i(\tilde{x}) = h_i(\tilde{x}) : f_{j_i}(\tilde{g}_i(\tilde{x})) \tag{2}$$

The functions $\tilde{h}$ are the *head-functions*, and $\tilde{g}_i$ $(i = 1..r)$ the *step-functions*.

If $p : \{1..m\} \to \{1..m\}$ is defined by $p(i) = j_i$, then Equation (2) can be written $f_{p(i)}(\tilde{g}_i(\tilde{x}))$. In most cases of interest p is a permutation, but it need not be. For $\tilde{h} : S^r \to \Sigma$, $\tilde{g}_i : S^r \to S^r$, and p as above we write $\mathbf{corec}^{r,m,p}[\tilde{h}, \tilde{g}_1, \ldots, \tilde{g}_m]$ for the function tuple $\tilde{f} : S^r \to S^r$ defined as above. We omit the indices r, m, p when in no danger of confusion.

Note that the schema (2) requires that each cycle generate an output symbol. This is natural, because the focus of coinductive computing is generating the output, rather than consuming the input as in the scheme of recurrence (i.e. primitive-recursion). Just as recurrence consumes one input element at each computation cycle, thereby guaranteeing termination of the computation with a finite output, the scheme of corecurrence generates one output element in each cycle, thereby guaranteeing a productive (i.e. infinite) output. We comment below on an alternative, "lazy," variant of corecurrence, where the production of an output symbol is optional.

3.2 Primitive corecursive stream functions

Definition 1. *The* initial stream-valued functions *are:*

- *Projections functions* $P_i^n : S^n \to S$ $(1 \le i \le n)$*, defined by* $P_i^n(x_1, \ldots, x_n) = x_i$.
- *The tail function* $tl : S \to S$*, defined by* $tl(a : \sigma) = \sigma$.
- *The branching function* $B : \Sigma \times S^\ell \to S$*, defined by* $B(a, y_1, \ldots, y_\ell) =$ **if** $a = a_i$ **then** y_i.

The initial symbol-valued functions *are:*

- *Projection functions* $Q_i^n : \Sigma^n \to \Sigma$ $(1 \le i \le n)$*, defined by* $Q_i^n(x_1, \ldots, x_n) = x_i$.
- *For each* $a \in \Sigma$ *a nullary function* a.
- *The head function* $hd : S \to \Sigma$.
- *The branching function* $L : \Sigma^{\ell+1} \to \Sigma$*, defined by* $L(a, z_1, \ldots, z_\ell) =$ **if** $a = a_i$ **then** z_i.

The primitive corecursive (p.c.) stream functions *are generated from the initial functions above using type-correct composition and the scheme of corecurrence 2. That is, a function over streams and letters is p.c. when it is the denotation of a term of the typed-lambda calculus with base types Σ and S, and constants for the initial functions and the corecursion operator above.*

3.3 Locality and continuity under corecurrence

Suppose

$$f(x_1 \ldots, x_r) = h(\tilde{x}) : f(g_1(\tilde{x}), \ldots, g_r(\tilde{x})) \tag{3}$$

with each g_i computed by a JFT G_i and h by H. In general we cannot expect f to be computed by a JFT: if it were, then it would be Turing-computable in logspace [14], but Example [14, §2.3(vi)] shows that it need not be.

However, our next Lemma shows that if the arguments $x_1, \ldots, x_r$ are such that G_j is weakly local on x_j and local on $x_{j+1}, \ldots, x_r$, then a finite amount of information about $\tilde{g}^{[n]}(\tilde{x})$,[4] of the same size for all n, permits the computation of $\tilde{g}^{[n+1]}(\tilde{x})$. This makes it possible to define a JFT which is continuous on all arguments.

Lemma 2. *Suppose $f = \mathbf{corec}[h, \tilde{g}, \tilde{y}]$, as in 3. Suppose the corecursive arguments fall into q disjoint sub-lists $\tilde{x}_i$, where $\tilde{x}_i = \langle x_{i1}, \ldots, x_{ir_i} \rangle$, and that g_{ij} is computed by a JFT G_{ij} and h is computed by a JFT H so that:*

1. *Each G_{ij} is continuous on $\tilde{x}_m$ for $m < i$, with modulus ω_{ij} of order $n^{d_{ij}}$;*
2. *each G_{ij} is weakly-local on arguments $\tilde{x}_i$ (with bound b_i);*
3. *each G_{ij} is local on $\tilde{x}_m$ where $m > i$ (with bound b_i);*
4. *all cursor jumps and* **Go** *operations are within the same group: when G_{ij} jumps a cursor or fires a* **Go**, *it must involve cursors on arguments of group i.*

Then f is computed by a JFT F which is continuous on all corecursive arguments, with as modulus a finite composition of depth $\leq r$ of the moduli ω_{im}.

Proof. Without loss of generality we assume that all b_i's are identical (and denoted b), and that $q = r$, with each group A_i consisting just of x_i. We can thus write G_i for G_{i1}. We also assume that $\omega_{i,i+p} = \omega_{i,i+1} \circ \omega_{i+1,i+2} \circ \cdots \circ \omega_{i+p-1,i+p}$, that is, the modulus of G_i on the $i+p$-th input is the composition of the one-step moduli. The proof of the general case only requires more tedious detail.

Define the *i-th cache* for a stream $\tilde{g}^{[n]}(\tilde{x})$ to be the list C_i of the first $\omega_{ir}(b)$ entries of $(\tilde{g}^{[n]}(\tilde{x}))_i$. The *cache* for $\tilde{g}[n]n(\tilde{x})$ is the list $C_1; \cdots; C_r$, $i = 1, \ldots, r$.

We now describe a JFT F that computes f, and is continuous on all its input. By assumption (4) of the Lemma, if an update of the cache invokes a **Go** by some G_i, then it must be on its i-th argument, and since G_i is weakly-local on that argument, it is invoked to generate a tail of x_i. When this operation is

[4] We write $\tilde{g}^{[n]}$ for the n-th iteration of $\tilde{g}$

iterated, it therefore remains true that a **Go** is triggered by G_i to generate a tail of x_i.

G_i is weakly-local on its i-th argument; so when applied to $\tilde{g}^{[n]}(\tilde{x})$, G_i triggers a **Go** on $(\tilde{g}^{[n]}(\tilde{x}))_i$. By induction on n it follows that every such **Go** yields a stream which is a tail of the original i-th input x_1. F will use a reserved cursor on x_i to record that position. We refer to that cursor as the *i-th pointer*.

F calculates successively for $n = 1, 2, ...$ the caches and pointers for $\tilde{g}^{[n]}(\tilde{x})$. Each cycle concludes with F invoking H to calculate $h(\tilde{g}^{[n]}(\tilde{x}))$: since H needs only the first b entries of each of its inputs, and $\omega_{iq}(n) \geq n$ for all i, H needs only use the cache for that calculation.

F starts by initializing the cache for i to the first $\omega_{ir}(b)$ entries of x_i, and the pointer for i to the head entry of x_i.

F then proceeds to calculate the caches and pointers for $\tilde{g}^{[n+1]}(\tilde{x})$, from the caches and pointers for $\tilde{g}^{[n]}(\tilde{x})$, as follows. To calculate the first ω_{ir} entries of $(\tilde{g}^{[n+1]}(\tilde{x}))_i$ F invokes G_i. This calculation uses up to $\omega_{ji}(b)$ entries of $(\tilde{g}^{[n]}(\tilde{x}))_j$ for $j < i$, and up to b entries of $(\tilde{g}^{[n]}(\tilde{x}))_j$ for $j \geq i$. These are all available in caches for $\tilde{g}^{[n]}(\tilde{x})$, by our assumptions about the moduli $\omega_{..}$. As noted, G_i finally triggers a **Go** on its i-th argument, namely $(\tilde{g}^{[n]}(\tilde{x}))_j$, by a cursor at position $p \leq b$. F thus steps its i-th pointer p times.

Clearly, the JFT F is continuous on its corecurrence arguments. □

4 Ramified corecurrence

4.1 Computing with ramified data

Ramified recurrence [2, 9] is based on a distinction between tiers (i.e. operational levels) of data. Inductive data of tiers t supports recurrence of (possibly parametrized) functions between data of lower tier. A consequence is that a function defined by recurrence on data of tier t has output of tier $< t$, thereby blocking the self-feeding of the output into the recurrence position, as would be the case in defining the iterate of a tier-lowering function g, $f(n) = g^{[[n]]}(\mathtt{0})$ by the recurrence $f(\mathtt{s}(x)) = g(f(x))$. Intuitively, the generative process leading up to the recurrence argument has a higher "energy level" than the function's output.

Dually, *ramified corecurrence,* requires that the output tier of a corecursive function is higher than the tiers of its inputs. Intuitively, this is because the decomposition of the output requires greater energy than the decomposition of the arguments. The duality between inductive and coinductive data is brought out in ramified second order logic [16]. Here one ramifies set variables, and restrict the instantiation of a universal set quantifiers: if X is a set variable of tier t, then formula $\forall X^t\ \varphi[X]$ can be instantiated to a set-definition $\lambda z.\varphi$ only when the formula φ does not refer to quantified set-variables of tier $\geq t$ or to free set variables of tier $> t$. In particular, this restriction blocks impredicative set-existence (Comprehension).

Referring in the ramified context to second-order definitions of inductive data using $\forall$ over sets, it follows that recurrence is admissible only if it is ramified.

Dually, given that coinductive data is definable using $\exists$ over sets, corecurrence is admissible only if it is ramified in the sense above. See e.g. [14] for a more detailed discussion.

4.2 Ramified corecurrence

To formally convey the notion of ramified corecurrence, we posit copies S_i ($i \geq 0$) of the set S of Σ-streams, dubbed *tiers.* We can construe these as disjoint base types, in addition to the base type Σ.

As usual, we give rules for typing judgments of the form $\Gamma \vdash e : \tau$, where Γ is a type environment, e an expression and τ a type. We write Γ, Γ' for the union $\Gamma \cup \Gamma'$, which is implicitly assumed to be legal (i.e. without multiple types assigned to the same variable). Arrows associate to the right. Since we do not deal with higher types, our type environments will refer only to arrow-free types. The type system is given in the following table. The main points to keep in mind are these:

1. Each stream comes with a *tier*, which conveys its computational strength as input for a corecursive definition. Symbols $a \in \Sigma$ are not classified into tiers.
2. Ramified corecurrence defines a function from inputs of various tiers, to an output at a tier that majorizes them all. As explained above, a higher tier means here weaker computation power, so ramified corecurrence degrades the computation power of its arguments, by using them.
3. Composition is typed as usual, which implies here that it respects tiers.

Type system RC for the primitive corecursive functions

Underlying alphabet: $\Sigma = \{a_0, a_1, \ldots, a_\ell\}$.

Generic lambda rules

$$\frac{}{\Gamma \vdash x : \tau} \; (x : \tau \text{ in } \Gamma)$$

$$\frac{\Gamma, x : \tau \vdash E : \sigma}{\Gamma \vdash \lambda x.E : \tau \to \sigma} \qquad \frac{\Gamma \vdash E : \tau \to \sigma \quad \Gamma' \vdash F : \tau}{\Gamma, \Gamma' \vdash E(F) : \sigma}$$

$$\frac{\Gamma_j \vdash E_j : \tau_j \;\; (j = 0, 1)}{\Gamma_0, \Gamma_1 \vdash \langle E_0, E_1 \rangle : \tau_0 \times \tau_1} \qquad \frac{\Gamma \vdash E : \tau_0 \times \tau_1}{\Gamma \vdash \pi_j(E) : \tau_j} \;\; (j = 0, 1)$$

Initial functions

$$\frac{}{\vdash P_k^n : S_{i_1} \times \cdots \times S_{i_n} \to S_{i_k}}$$

$$\frac{}{\vdash \varepsilon : S_j}$$

$$\frac{}{\vdash tl : S_j \to S_j}$$

$$\frac{}{\vdash B : \Sigma \times S_j^\ell \to S_j}$$

$$\frac{}{\vdash a : \Sigma} \qquad (a \in \Sigma)$$

$$\frac{}{\vdash hd : S_j \to \Sigma}$$

$$\frac{}{\vdash L : \Sigma^{\ell+1} \to \Sigma}$$

Corecurrence

If τ is a product of r base types,
and is of tier $i > 0$ where $i < j$, then

$$\frac{\Gamma \vdash h : \tau \to \Sigma^m \quad \Gamma' \vdash g : (\tau \to \tau)^m}{\Gamma, \Gamma' \vdash \mathbf{corec}^{r,m,p}[h, g] : \tau \to S_j^m}$$

4.3 The computational contents of ramified corecurrence

Theorem 2. *If a primitive-corecursive function f is defined by a term $\mathbf{t} : S_{i_1} \times \cdots \times S_{i_r} \to S_j$ in the type system* **RT***, then f is computable by a JFT F. Moreover, F is local on each argument of tier $> j$, weakly-local on each argument of tier S_j, and continuous with a polynomial modulus on each argument of tier $< j$.*

Also, if a symbol-valued function is definable by a term $\mathbf{t} : S_{i_1} \times \cdots \times S_{i_r} \to \Sigma$, then it is local on all arguments.

Proof. The proof proceeds by induction on the typing derivation of $\mathbf{t}$ in **RT**. The base cases are straightforward. The induction step for composition is given by Lemma 1, and the step for ramified corecurrence by Lemma 2. □

4.4 Characterization of ramified corecurrence

The schema (1) requires that each computation cycle generate an output symbol. The schema of *lazy corecurrence* relaxes this requirement, using *test functions* k_i to indicate whether or not it is triggered:

$$f_i(\tilde{x}) = \begin{cases} f_i(\tilde{x}) & \text{if } hd(k(\tilde{x})) = \varepsilon \\ f_j(\tilde{g}_i(\tilde{x})) & \text{otherwise} \end{cases} \tag{4}$$

This relaxation is useful for simulation of machine models, such as JFT, where computation steps need not always emit an output symbol. Indeed,

Theorem 3. [14, Proposition 4.1] *If a function over streams is computed by a JFT then it is definable from the initial functions using composition and lazy corecurrence.*

However, lazy corecurrence goes counter the very rationale of corecurrence. The point of the schema of recurrence (i.e. primitive recursion) is that it consumes its input, thereby guaranteeing termination. Dually, the point of corecurrence is that it builds its output, thereby guaranteeing productiveness. We might, for instance, consider a schema of "lazy recurrence", where the consumption of the input is optional; the result would simply be equational programs in the style of Herbrand-Gödel [8], i.e. a Turing-complete computation model. Lazy corecurrence has a similar effect for computing over streams: it captures Turing computability over functions of type $\mathbb{N} \to \Sigma$.

A drastic ramification of the definition into two tiers, reduces the computational power to that of a JFT [14], but the class of functions definable this way is not closed under composition.

Lazy corecurrence can be represented by our standard ("strict") corecurrence, by fixing a reserved symbol, say $\mathtt{e}$, to stand for the "empty" output-symbol of a lazy-corecurrence. The intended output stream σ is then obtained from the actual output τ by a *collapse* operation, that erases all occurrences of $\mathtt{e}$ in τ. This representation is less trifling than may first seem: it keeps a record of computational resources, which is useful for function composition.

Theorem 4. *Let f be a function over streams. The following are equivalent.*

1. *f is definable by 2-tier ramified lazy corecurrence.*
2. *f is the collapse of a function definable by (all tier) ramified corecurrence.*
3. *f is computable by a JFT.*

Proof. (1) implies (2) trivially.

(2) implies (3): Suppose f is the collapse of a function g defined by ramified corecurrence. By Theorem 2 g is computed by some JFT M, and so f is computed by the composition of M with a finite state transducer that erases all occurrences of $\mathtt{e}$. By Theorem 1 it follows that f is computed by a JFT.

(3) implies (1) by [14]. □

5 Log-space computation over finite streams

5.1 Computing over finite and regular streams

We have commented in the introduction about the practical significance of computing coinductively on very large (but finite) data. Several representations of finite streams are possible; the best suited to our purpose is the use of a symbol $\mathtt{e}$ as an end-marker. That is, a word $w \in \Sigma^*$ is represented by the stream that extends w with an indefinite

repetition of $\mathtt{e}$. (Recall that $\mathtt{e} \notin \Sigma$.)[5] We refer to w as the *significant portion* of the stream $w : \mathtt{e}^\omega \equiv w : \mathtt{e} : \mathtt{e} : \cdots$ representing w.

This representation of finite streams also agrees with a natural representation of *regular streams,* in the sense of [5], i.e. eventually-periodic streams. The salient property of such streams is that they have only finitely many distinct sub-streams.

Of course, we are particularly interested in the finite streams. Given a function $f : S^r \to S$ let f^{fin} denote the partial-function on Σ^* obtained by restricting f to finite streams, that is:

$$f^{fin}(\tilde{w}) = \text{the significant portion of the collapse of } f(\tilde{w})$$

Proposition 1. *If a function $f : S^r \to S$ is computed by a JFT M, then it maps (vectors of) regular streams to regular streams.*

More precisely, suppose that M has d states and k cursors. If $\tilde{\sigma} = (\sigma_1, \ldots, \sigma_r)$ is such that each σ_i has at most n distinct sub-streams, then $f(\tilde{\sigma})$ has at most $d \cdot n^{rk}$ distinct sub-streams.

Proof. The number of possible configurations of M for input $\tilde{\sigma}$ is $d \cdot n^{rk}$. □

Virtually the same argument establishes:

Corollary 1. *If $f : S^r \to S$ is computed by a JFT M, then f^{fin} is computable in logarithmic space.* □

5.2 Simulation of log-space by JFTs

In [14] we considered Turing transducers over streams. Such transducers are similar to JFT's, except that they have an auxiliary read/write memory. Note that we allow such machines to have several cursors on each input stream. We say that a Turing stream-transducer is *log-space* if the work-tape is restricted to size $O(\log n)$ when computing the n'the entry of the *output.*

A well-known simulation of log-space computation by multi-cursor two-way automata [7] is based on the representation of configurations of log-space size by cursors on the input. Our Turing transducers on streams can similarly be simulated by JFT's, provided the latter can place cursors on some word of length n when calculating the n'th output entry. That proviso is no longer needed when the inputs are finite streams. Indeed, if $f : S^r \to S$ is computed by a logspace Turing stream-transducer, then there is a JFT G which computes a function g that agrees with f on all finite input. Here we are interested in the following restricted case of that observation:

Proposition 2. *If $g : (\Sigma^*)^r \to \Sigma^*$ is computable by a log-space Turing transducer M, then there is a JFT F computing a function $f : S^r \to S$ such that $g = f^{fin}$.*

Proof. F simulates the work-tape of M using additional cursors on (the significant portion of) the input, whose end is detected by M by the first occurrence of $\mathtt{e}$. Once M completes emitting output symbols for the given inputs, F proceeds to emit $\mathtt{e}$ indefinitely. □

Combining Corollary 1, Proposition 2, and Theorem 4 we obtain

[5] A more principled representation uses of a nullary constructor for the empty word, yielding a coinductive type for both finite and infinite streams. That would call, though, for a restatement of the corecurrence schemas, and would not mesh well with the regular streams discussed below.

Theorem 5. *Let $g : (\Sigma^*)^r \to \Sigma$. The following are equivalent:*

1. *g is log-space computable.*
2. *$g = f^{fin}$ for some $f : S^r \to S$ which is computable by a JFT, or equivalently definable by ramified lazy corecurrence, or equivalently is the collapse of a function definable by corecurrence.*

□

6 Conclusion and research directions

This paper is a contribution to the broad project of using ramification methods in Implicit Computational Complexity. We have settled here the status of the computational contents of ramified corecurrence in all tiers, and showed that it is captured by jumping finite transducers, and for finite streams by log-space Turing transducers. (The case of two tiers only was settled in [14] using the very strong limitations imposed on two tier corecurrence, due to the fact that the step functions must in that case be unary.) The converse holds, in a slightly modified form, by our previous work [14].

The methods developed here seem promising in tackling several more general or related questions. An obvious one is the generic extension of our treatment to arbitrary coinductive data, such as finite/infinite trees. We would expect that the extension of JFTs to such data,[6] might answer this question. Moreover, we would like to characterize the computational nature of ramified computing over data generated by both inductive and coinductive, for example in the framework of [12].

The effect of ramification in such a broad context can be brought to bear on more general forms of recurrence and corecurrence, in particular using higher type functionals. In [10] it was shown that the functions over inductive data defined by ramified recurrence in all finite types are precisely the Kalmar-elementary functions (as opposed to the ε_0-recursive functions in the un-ramified case), and [11] shows that the type-2 functionals defined by ramified recurrence (on base type) form precisely the Cook-Urquhart's class **BFF** of Basic Feasible Functionals. Our aim is to explore such results in the broader context of inductive/coinductive data-types.

Closer to home, we would like to better understand the nature of computational complexity over coinductive types in general, and streams in particular, e.g. the status of the "size yardsticks" that we used in the JFT simulation of logspace stream-transducers in [14].

Last but not least, there are intriguing and promising questions on the use of ramified corecurrence in the analysis and understanding of practical algorithms on streams and other coinductive data.

References

1. Stephen Bellantoni. *Predicative recursion and computational complexity.* PhD thesis, University of Toronto, 1992.
2. Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the poly-time functions, 1992.

[6] Tree automata have been studied extensively [3]; jumping automata on *finite* graphs, i.e. regular trees, were considered in [4].

3. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007.
4. Stephen A. Cook and Charles Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM J. Comput.*, 9(3):636–652, 1980.
5. Guy Cousineau. An algebraic definition for control structures. *Theoretical Computer Science*, 12:175–192, 1980.
6. Artur Czumaj, S. Muthu Muthukrishnan, Ronitt Rubinfeld, and Christian Sohler, editors. *Sublinear Algorithms, 17.07. - 22.07.2005*, volume 05291 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
7. Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Inf.*, 1:336–344, 1972.
8. Stephen C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhof, Groningen, 1952.
9. 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.
10. Daniel Leivant. Ramified recurrence and computational complexity III: higher type recurrence and elementary complexity. *Ann. Pure Appl. Logic*, 96(1-3):209–229, 1999.
11. Daniel Leivant. Intrinsic reasoning about functional programs I: first order theories. *Ann. Pure Appl. Logic*, 114(1-3):117–153, 2002.
12. Daniel Leivant. Global semantic typing for inductive and coinductive computing. *Logical Methods in Computer Science*, 10(4), 2014.
13. 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.
14. Ramyaa Ramyaa and Daniel Leivant. Ramified corecurrence and logspace. *Electronic Notes in Theorerical Computer Science*, 276:247–261, 2011.
15. Ronitt Rubinfeld and Asaf Shapira. Sublinear time algorithms. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:13, 2011.
16. Kurt Schütte. *Proof Theory*. Springer-Verlag, Berlin, 1977.