UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

May 22, 2025

Efficient Coinductives through State-Machine Corecursors

William Sørensen

Gonville & Caius College

Submitted in partial fulfilment of the requirements for the
Computer Science Tripos, Part II

Declaration

I, William Sørensen of Gonville & Caius College, being a candidate for the course, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.
Signed: William Sørensen
Date: May 22, 2025

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit.

Contents

1 Introduction 7
1.1 Dependent type theory 7
1.2 Polynomial functors 8
1.2.1 Common pfunctors 9
1.2.2 Lean formalization 11
1.2.3 F-(co)algebras 11
2 Preparation 12
2.1 Induced Productivity Transform 12
3 Implementation 13
3.1 The ABI Type 13
3.1.1 Weak univalence 14
3.1.2 Relation to Shrink and further universe transforming types 14
3.2 Stream implementation 14
3.3 Expanding the progressive approximation theory 14
3.3.1 Universe lifting of polynomial functors. 15
3.3.2 Generalizing the corecursor 15
3.4 State machine encoding 15
3.4.1 PreM 15
3.5 Interaction trees 15
4 Evaluation 16
4.1 Performance between SME and PA 16
5 Conclusions 18
Bibliography 19
6 Appendicies 20

Word count: 1515

Chapter 1

Introduction

1.1 Dependent type theory

1.2 Polynomial functors

A (multivariate) polynomial functor on set is: a head-type, along with a (collection of) child family(ies) parameterised by the head. An object of a polynomial functor is a select head type, and the corresponding child(ren) parameterised by the head as seen in Equation 1 (this shows only the monovariate version)[1]. Moraly polynomial functors can be thought of as types generic in some set of arguments, with a collection of constructors (the head type), where the children correspond to how many of the polymorphic argument are wanted.

$$(h : H) \times c_h \to \alpha \equiv \sum_{h:H} \alpha^{c_h} \tag{1}$$

I find this notation a bit difficult to read, so for a multivariate polynomial functor $P\alpha_0...\alpha_i$, with head types as variants $h_0...h_j$, and children c_{i,h_j}, it would conventionally be written as Equation 2, I will display the hs and cs as in Figure 1. I write an object of this functor, picked at h_0 with functions f_i, as seen in Figure 2. The justification of this notation will be seen Section 1.2.2.

$$(j' : \text{Fin } j) \times \left(h : h_{j'}\right) \times (i' : \text{Fin } i) \to c_{i',h} \to \alpha_{i'} \tag{2}$$

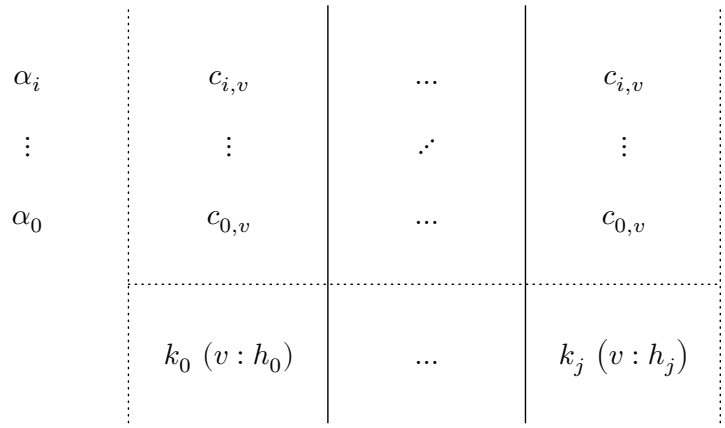

Figure 1: The head and child components of $P\alpha_0...\alpha_i$

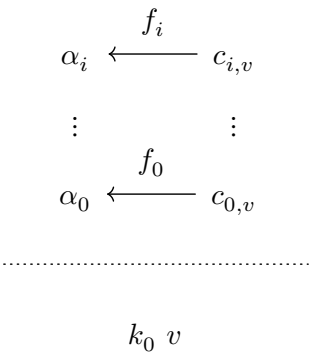

Figure 2: An object of $P\alpha_0...\alpha_i$

Mapping on MvPfunctors is done by composition of the arrows. This can be seen in Figure 3. By set forming a category, mapping a composition is the same as mapping each one by one.

[1]More can be read on the ncatlab article (nLab authors, 2026), we restrict ourselves to Set, so I recommend skipping to that section.

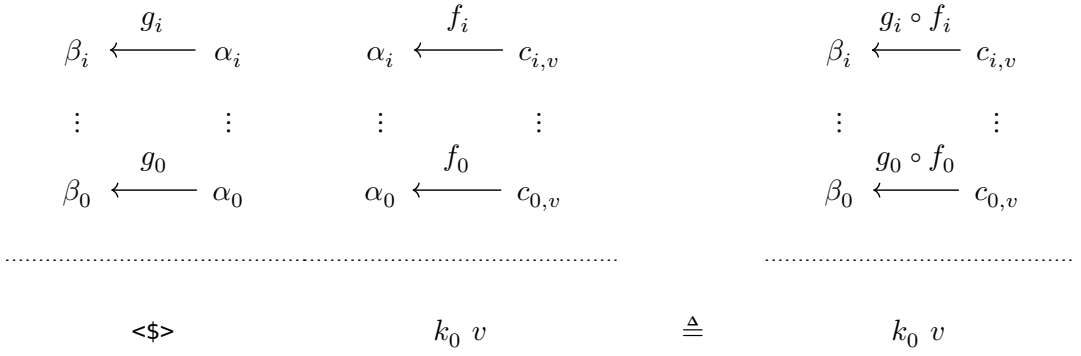

Figure 3: Definition of mapping of mvpfunctors,

in this case specalised to $P\alpha_0...\alpha_i$

1.2.1 Common pfunctors

1.2.1.1 Constant

A functor, fixed at a constant type T and a arity i, denoted $\text{const}_i T$, shown in Figure 4. It has one constructor $\text{mk}_\alpha : T \to \text{const}_i T\alpha_0...\alpha_i$, with an inverse $\text{get}_\alpha : \text{const}_i T\alpha_0...\alpha_i \to T$.

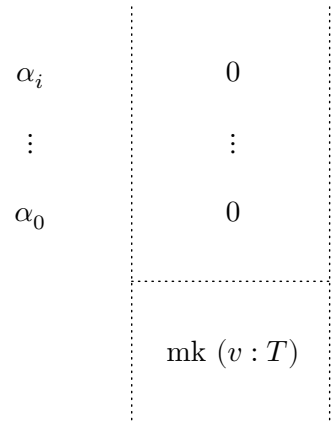

Figure 4: Definition of $\text{const}_i T$

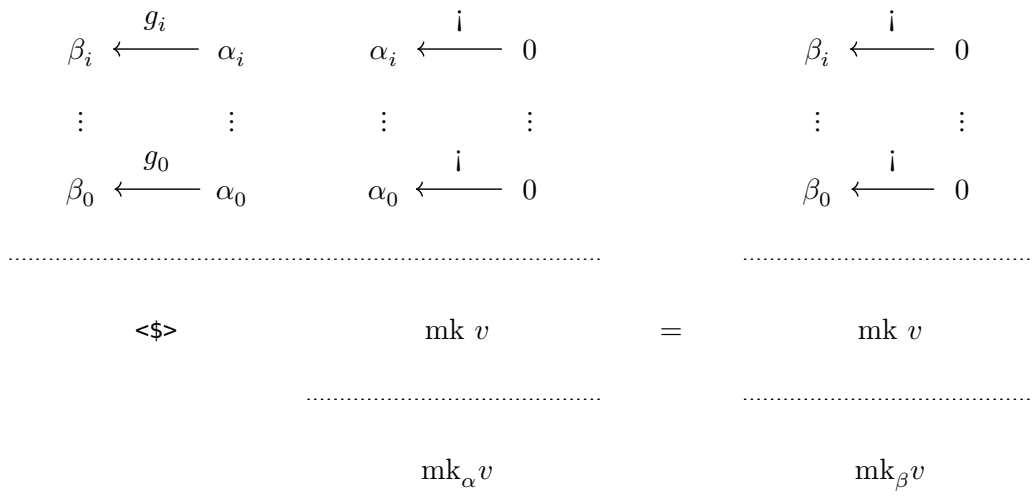

Figure 5: By initial hom uniqueness,

the object is invariant under mapping.

1.2.1.2 Prj

Given a typevector $\alpha_0 ... \alpha_i$, and a select index into the vector ($n : \text{Fin } i$), the projection functor holds a value of type α_n. It is defined in Figure 6. It has one constructor $\text{mk}_\alpha : \alpha_n \to \text{prj}_n \alpha_0 ... \alpha_i$, with an inverse $\text{get}_\alpha : \text{prj}_n \alpha_0 ... \alpha_i \to \alpha_n$.

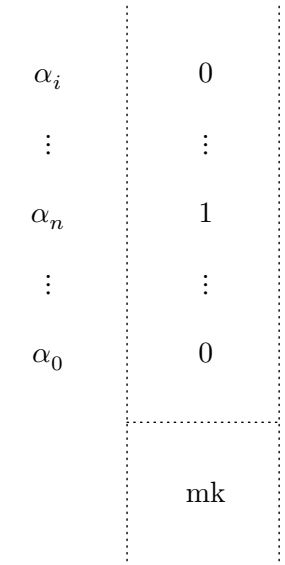

Figure 6: Definition of prj_n

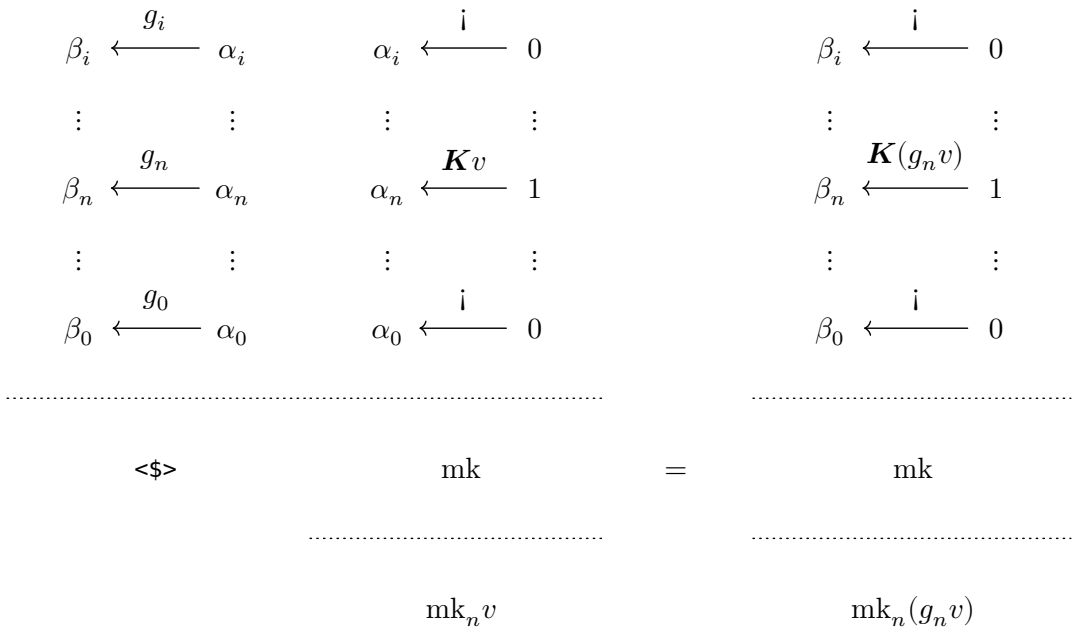

Figure 7: By initial hom uniqueness,

mapping only changes the nth argument.

1.2.1.3 Sum

1.2.1.4 Option

1.2.1.5 Functions from fixed types

1.2.1.6 W

1.2.1.7 M

1.2.2 Lean formalization

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

1.2.3 F-(co)algebras

Chapter 2

Preparation

2.1 Induced Productivity Transform

During my internship under AK and TG, I made a structure I then called `DeepThunks`. I now refer to them as the induced productivity monad. They are the general way of constructing productive functions on cofixed points from terminating functions.

Given a polynomial $\boldsymbol{P}\bar{\alpha}\beta$, then the corecursor we get for $\mathbf{M}\ \boldsymbol{P}\bar{\alpha}$ is:

$$\text{corec} : \{\beta\} \to (\beta \to \boldsymbol{P}\bar{\alpha}\beta) \to \beta \to \mathbf{M}\ \boldsymbol{P}\bar{\alpha} \tag{3}$$

Often this is adequet, but one has a few drawbacks that can be summarised as: it has to emit exacly one 'layer' of the final structure. The end goal would be to allow it to emit anything from layer 1 to the entire structure. One can think of this as taking the most general choice of β. The structure that solves this is $\mathbf{PT}\ \boldsymbol{P}\bar{\alpha}\beta \triangleq \mathbf{M}_{\xi}\boldsymbol{P}\bar{\alpha}(\beta \oplus \xi)$. From here we construct two unique functions, inject and dtcorec:

$$\text{inject} : \{\beta\} \to \mathbf{M}\ \boldsymbol{P}\bar{\alpha} \to \mathbf{PT}\ \boldsymbol{P}\bar{\alpha}\beta \coloneqq \text{corec}_{\mathbf{PT}\ \boldsymbol{P}}(\text{map}_{\mathbf{M}\ \boldsymbol{P}}(\mathbb{1} ::: \iota_r) \circ \text{dest}) \tag{4}$$

$$\text{dtcorec}\ \{\beta\}(g : \beta \to \mathbf{PT}\ \boldsymbol{P}\bar{\alpha}\beta) : \beta \to \mathbf{M}\ \boldsymbol{P}\bar{\alpha} \coloneqq \text{corec}_{\mathbf{M}\ \boldsymbol{P}}((\text{map}_{\mathbf{M}\ \boldsymbol{P}}(\mathbb{1} ::: [g, \mathbb{1}])) \circ \text{dest}) \circ g \tag{5}$$

Chapter 3

Implementation

3.1 The ABI Type

The problem the ABI type tries to tackle is one of abstracting the runtime datatype through functions. Given an isomorphism eq : $\alpha \cong \beta$ for some types α and β, my first try at solving this involved constructing an object ABI $\alpha\beta$, making the following commute:

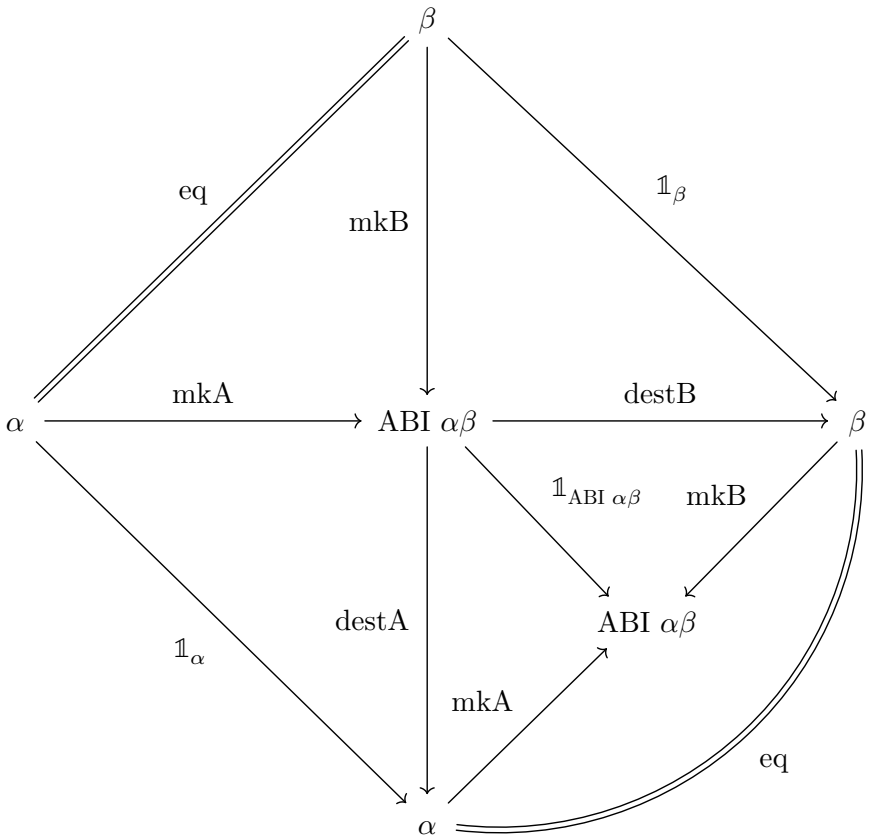

Figure 8: Operations on ABI

Additionally I had an elimination principle satisfying the two equations below.

```
1  def elim : {motive : ABI _ _ eq → Type w}                    Lean
2         → (hLog : (z : A) → motive (mkA z))
```

```
3         → (hCheap : (z : B) → motive (mkB z))
4         → (eqA : ∀ z, hLog z ≍ hCheap (eq z))
5         → (eqB : ∀ z, hCheap z ≍ hLog (eq.symm z))
6         → (v : ABI _ _ eq) → motive v := _
7
8   theorem elimLog : {motive : carry → Type w}
9         → {hLog : (z : A) → motive (mkA z)}
10        → {hCheap : (z : B) → motive (mkB z)}
11        → {eqA : ∀ z, hLog z ≍ hCheap (eq z)}
12        → {eqB : ∀ z, hCheap z ≍ hLog (eq.symm z)}
13        → ∀ z, elim hLog hCheap eqA eqB (mkA z) = (hLog z) := _
14  theorem elimCheap : {motive : carry → Type w}
15        → {hLog : (z : A) → motive (mkA z)}
16        → {hCheap : (z : B) → motive (mkB z)}
17        → {eqA : ∀ z, hLog z ≍ hCheap (eq z)}
18        → {eqB : ∀ z, hCheap z ≍ hLog (eq.symm z)}
19        → ∀ z : B, elim hLog hCheap eqA eqB (mkB z) = (hCheap z) := _
```

Through quite a bit of work (which I call transliteration, as seen in the ABI file), You can free a universe level and open for a more general usage of the function.

3.1.1 Weak univalence

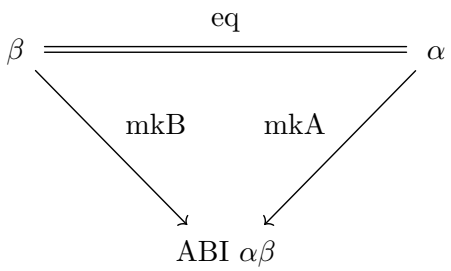

Figure 9: Weak univalence up to shrink

3.1.2 Relation to Shrink and further universe transforming types

3.2 Stream implementation

3.3 Expanding the progressive approximation theory

During the pheasability assesment I noticed that, in the current formalised theory of polynomials, the statement wouldn't even type-check. This stemmed from a problem with the corecursive principle for the M type in the old implementation. corec : $\{\alpha : \text{TypeVec}.\{\mathcal{U}\}n\} \to \{\beta : \text{Type } \mathcal{U}\} \to (g : \beta \to P(\alpha ::: \beta)) \to \beta \to M\alpha$[2]. The problem here is that both α and β have to both reside in $\mathcal{U}$. Solving this is done through the next two sections.

[2]https://github.com/leanprover-community/mathlib4/blob/7a60b315c7441b56020c4948c4be7b54c222247b/Mathlib/Data/PFunctor/Multivariate/M.lean#L152-L154

3.3.1 Universe lifting of polynomial functors.

The main problem caused here comes from the fact that lean isnt cummulative. This means it is impossible to express a universe hetrogouns typevector. In other words $\alpha ::: \beta$ is only typable if $\alpha : \text{TypeVec}.\{\mathcal{U}\}n$ and $\beta : \text{Type}\ \mathcal{U}$. The natural way of solving this is using the supremum in universe levels you get from $\text{ULift} : \text{Type}\ \mathcal{U} \to \text{Type}\ (\max \mathcal{U}\mathcal{V})$. This means we can have $\beta : \text{Type}\ \mathcal{U}$ and $\alpha : \text{Type}\ \mathcal{V}$, then ulift both of them to a common universe $\text{ULift}\ \alpha ::: \text{ULift}\ \beta : \text{TypeVec}.\{\max \mathcal{U}\mathcal{V}\}(n+1)$[3].

Noticable the next hurdle we encounter is that PFunctors are restricted to a universe level. Recall the definition from Section 1.2.2. Observe how for a $\text{MvPfunctor}.\{\mathcal{U}\}n$, we require that both the head and child reside in $\mathcal{U}$. This will also cause problems, as looking back at the definition of the corecursor, we will require P to be able to accept $\text{ULift}\ \alpha ::: \text{ULift}\ \beta$. If we do not add the ability to lift P, the unifier will force $\mathcal{U} = \mathcal{V}$, thereby invalidating all the work we did in the previous section. Luckily lifting a PFunctor is relatively easy. We define it as $\text{ULift}\ P \triangleq \langle \text{ULift}\ P.1, \lambda x \mapsto \text{ULift}\ (P.2x) \rangle$. This works and now we can move on to our goal[4].

3.3.2 Generalizing the corecursor

Now with all the work in the previous section, by generalizing corec'[5], we can define $\text{corecU} : \{\alpha : \text{TypeVec}.\{\mathcal{U}\}n\} \to \{\beta : \text{Type}\ \mathcal{V}\} \to (g : \beta \to \text{ULift}\ P(\text{ULift}\ \alpha ::: \text{ULift}\ \beta)) \to \beta \to M.\{\mathcal{U}\}\alpha$. Notably we are able to fit the object into $\mathcal{U}$ (as opposed to in the SME).

The expected diagram using corecU and dest commutes.

3.4 State machine encoding

Noting the definition of corecU, one might wonder if you could define M from first principles for this. The problem one encounters is one of universes. As seen in the definition above, if one were to define a type whos construcor is directly the corecU definiton, it would hold a $\beta : \text{Type}\ \mathcal{V}$. This then forces the object to reside in $\text{Type}\max \mathcal{U}(\mathcal{V}+1)$. This is a problem as one loses most closure results as you will be lifting more and more. The main beinifit from this is the performance aspect. Going from reading to a depth n, to a depth $n+1$ is not $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$. This will be seen in Section 4.1. We will henceforth refer to the datatype SME.PreM.

3.4.1 PreM

As we speak about in Section 1.2.3, the M Type is the terminal object in the category of coalgebras. We can see through reasoning (cumilatively) in this category that PreM is weakly terminal. Looking at this category we want to somehow force the incoming morphisms together. This corresponds exactly to quotienting, for this we will use Bisimilarity.

3.5 Interaction trees

Interaction trees (ITrees) are a coinductive datastructure detailed in (Xia et al., 2019).

[3]Note we overload ULift as a notation to refer to lifting TypeVecs as well

[4]TODO: Speak with JV / W to see if this might be done in the lit, Ex : Locally presentable and accessable categories Adameck roshiski

[5]Done in PR NUMBER

Chapter 4

Evaluation

4.1 Performance between SME and PA

After building the equivelence, and instantiating shrink, we now have the ability to compare the performance between 3 representations: The SME encoding (high performance, hpRuns), the PA encoding (slow, slRuns), and a hand made big implementation (bigRuns). We compare these 3 in Figure 10. In Figure 10 we see the 3 representations specalised to infinite streams, the code for which can be seen in Listing 1

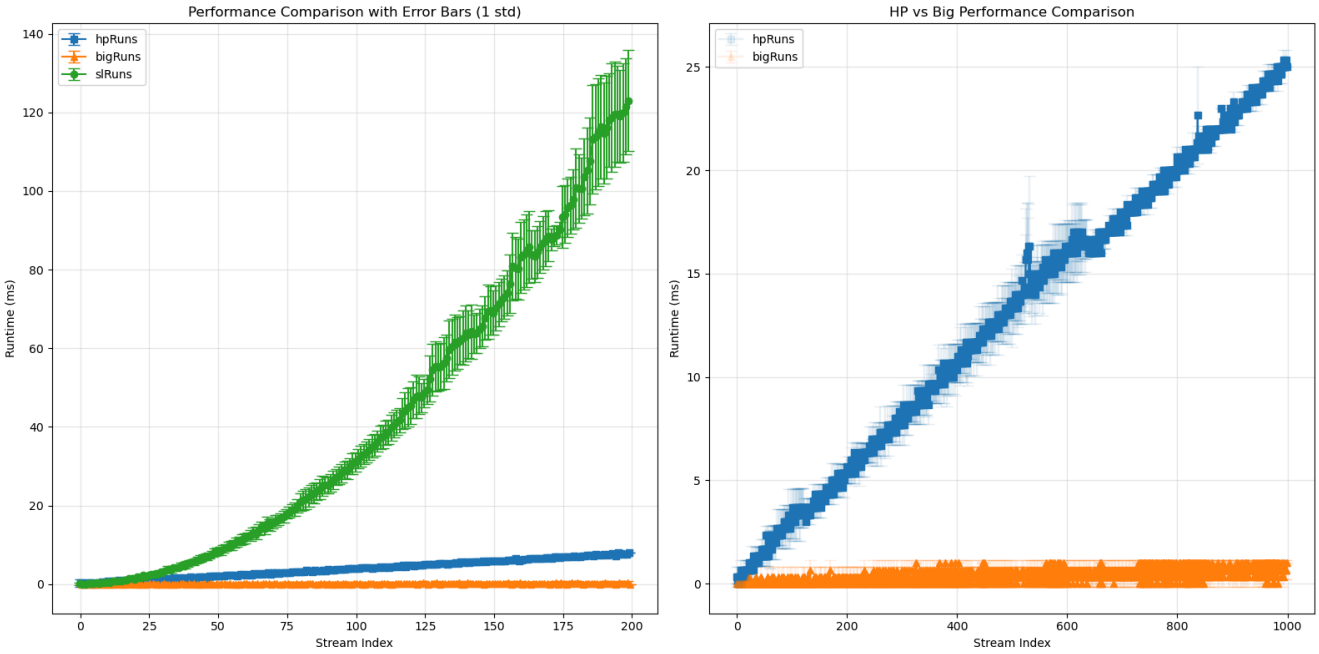

Figure 10: Performance of SME, PA, and Big representation

```lean
def QStream.Base : MvPFunctor 2 := ⟨
  Unit,
  fun _ _ => Unit
⟩

def QStreamSl α := M QStream.Base (fun _ => α)
def QStreamHp α := HpLuM QStream.Base (fun _ => α)

structure QStreamBig.{u} (α : Type _) where
  corec ::
    {t : Type u}
    (functor : t → Nat × t)
    (obj : t)

def numsSl : QStreamSl Nat :=
  .corec _ (fun n => ⟨.unit, fun | .fz, .unit => n.succ | .fs .fz, .unit => n⟩)
  Nat.zero

def numsHp : QStreamHp Nat :=
  .corec' (fun n => ⟨.unit, fun | .fz, .unit => n.succ | .fs .fz, .unit => n⟩)
  Nat.zero

def numsBig : QStreamBig Nat :=
  QStreamBig.corec (fun n => ⟨n, n + 1⟩) 0

def QStreamBig.getNth (x : QStreamBig Nat) : Nat → Nat
  | 0 => x.dest.fst
  | n+1 => x.dest.snd.getNth n

def QStreamSl.getNth (x : QStreamSl Nat) : Nat → Nat
  | 0 => match x.dest with
    | ⟨.unit, v⟩ => v (.fs .fz) .unit
  | n+1 => match x.dest with
    | ⟨.unit, v⟩ => QStreamSl.getNth (v .fz .unit) n

def QStreamHp.getNth (x : QStreamHp Nat) : Nat → Nat
  | 0 => match x.dest with
    | ⟨.unit, v⟩ => v (.fs .fz) .unit
  | n+1 => match x.dest with
    | ⟨.unit, v⟩ => QStreamHp.getNth (v .fz .unit) n
```

Chapter 5

Conclusions

Bibliography

nLab authors. (2026, January). *polynomial functor*.

Xia, L.-y., Zakowski, Y., He, P., Hur, C.-K., Malecha, G., Pierce, B. C., & Zdancewic, S. (2019). Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, *4*(POPL). https://doi.org/10.1145/3371119

Chapter 6

Appendicies