Safe Recursion Revisited:
Categorical Semantics and Type Systems for Lower Complexity

Mike Burrell
Computer Science Department
University of Western Ontario
London, Ontario, Canada, N6A 5B7

Robin Cockett Brian Redmond
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada, T2N 1N4

May 28, 2010

Abstract

The paper describes the categorical semantics and type theory underlying a polynomial time programming system called Pola. Pola is a reasonably expressive language which is functional in style. It allows the declaration of polarized inductive data types (with safe recursion), and of coinductive data. It has a polymorphic type system which ensures that every well-typed program has its running time bounded by a polynomial.

While the main purpose of the paper is to prove that the language is sound and complete for polynomial time programming, the proof that this is so involves a broader exploration of the categorical semantics and type theories for lower complexities. In particular, in the course of journey implicit type systems for PTIME (polynomial time), PSPACE (polynomial space), and ELEMENTARY (elementary time) are discussed.

The paper introduces, corresponding to the categorical semantics, a series of polarized type systems, on which the detailed bound calculations are performed. There are two systems which are discussed in detail: simply typed Pola and bunched Pola. Polarized type systems separate the world into opponent and player types: the simple system has an affine (closed) player world, polarized inductive data, and coinductive data and we show that this system is polynomial time sound and complete. Unfortunately, it is also well-known that this system is not very expressive. This leads to the development of a system which has a more sophisticated affine "bunched" type system for the player world. The bunched system is, in fact, *too* expressive: it is PSPACE sound and complete. However, we show how it can be cut down to PTIME by controlling the use of universal, coinductive, and higher-order types in the bunched contexts. This allows one to guarantee polynomial time computation while retaining a significant gain in expressive power.

The fact that the setting introduced here supports coinductive data distinguishes it quite sharply from traditional settings for PTIME. In particular, the presence of coinductive data (which must be *lazily* evaluated) allows one to show that P $\neq$ NP as these types allow access to distinguishing computations. The fact that setting has an implementation as a reasonably expressive programming language, suggests it is a reasonable model of PTIME computation worthy of further study.

Contents

1 Introduction **3**

2 The basic categorical setting **6**
2.1 Polarized strong categories . . . 6
2.2 Fibrational interpretation . . . 7
2.3 The category of $\mathcal{R}$-sized-sets . . . 10
2.4 Affine structure, products and coproducts in $\mathbb{X}$-strong categories . . . 11
2.5 The fibrational interpretation . . . 12
2.6 The interpretation in $\mathcal{R}$-sized sets . . . 13

3 Lifting, and comprehended recursion **16**
3.1 Lifting in $\mathbb{X}$-strong categories . . . 16
3.2 Lifting in fibrations: comprehension . . . 18
3.3 Trivial lifts and lifting in $\mathcal{R}$-sized sets . . . 19
3.4 Polarized operators . . . 20
3.5 Inductive recursion principles . . . 21
3.6 Counting the leaves of a tree . . . 23
3.7 Circular recursion in $\mathcal{R}$-sized sets . . . 24
3.8 Coinductive recursion principle . . . 25
3.9 Infinite lists . . . 26

4 Simply typed Pola **28**
4.1 Inductive data . . . 28
4.2 Operational semantics . . . 30
4.3 PTIME soundness . . . 32
4.4 PTIME completeness . . . 37
4.5 Coinductive types . . . 38
4.6 PTIME soundness with coinductive types . . . 40

5 Bunched Pola **43**
5.1 PSPACE completeness and QSAT . . . 43
5.2 Distributivity . . . 45
5.3 The bunched type system . . . 45
5.4 PSPACE soundness . . . 47
5.5 ELEMENTARY soundness and completeness . . . 52

6 PTIME within PSPACE **54**
6.1 Nondeterminism . . . 54
6.2 The polynomial hierarchy in Pola . . . 57
6.3 Cutting down to PTIME . . . 58

7 Conclusion **60**

1 Introduction

It is well-known that having a (strong) natural number object in a monoidal category immediately delivers all primitive recursive functions [21]. Therefore, to obtain settings which realize lower complexities something quite drastic has to be done.

An attractive feature, however, of a natural number object is that, as an initial data type, it arrives packaged with a universal property which enforces certain basic equalities on the maps involving that type. In the initial models of a doctrine involving such a type, these equalities become the basis for generating *all* the equality judgements. As computation is often viewed as arising from initial settings, this native notion of equality is of considerable interest.

In dealing with settings with complexity below primitive recursive – here referred to as *lower complexity* systems – our interest is not merely in the presence or absence of maps but also in the notion of equality which they support. Therefore, we would like data, even in these settings, to deliver a universal property and whence a native notion of equality.

This paper starts by describing how initial inductive data – together with their corresponding universal property – can be organized to express lower complexity settings. This development uses polarized strong categories, see [5], and the inductive and coinductive recursion principles they support. It then moves to the type theories for these systems in order to establish a series of bounding arguments which ultimately show how these systems can be used to express PTIME (polynomial time), PSPACE (polynomial space), and ELEMENTARY (elementary time).

This development grew from the realization that the system of Bellantoni and Cook [2] for describing PTIME had an immediate interpretation as a proof theory for a polarized logic. Polarities were introduced by Girard [8] to classify the behavior of the logical connectives in his "constructive" classical logic LC – an idea directly related to Andreoli's notion of *focusing* [1]. Olivier Laurent [16] further developed these ideas and quickly realized that there was a compelling connection to games [17]: these and further references to related developments are described in [9].

The general categorical proof theory for polarized logics and games is described in [6] and uses the notion of a polarized category. A polarized category is simply a module[1], however, viewed as a categorical structure in its own right. Polarization is produced by the separation beween the category which is the domain of the module (the "opponent" world) and the category which is the codomain (the "player world"). While Bellantoni and Cook used the terms "normal" and "safe" (respectively) for the two worlds of computation – rather than the game theory inspired terminology used here – it was clear that they were employing the technique of polarizing to achieve separation for computation.

Bellantoni and Cook's system of safe recursion, which only considered binary natural numbers, was a simplification of a slightly earlier system developed by Leivant [18] which had general inductive data types and infinitely many tiers (although two sufficed). Both these systems allowed duplication in their "safe" worlds and focused on controlling the recursion. The categorical doctrines and their type theories we present here, however, use a further crucial idea introduced by Hofmann [12]. He realized that it was advantageous to assume that the player (or safe) world was affine – so one cannot duplicate data. This allows one to restrict the "safe" or player world to *constant time* computations (in context) and this makes it quite clear, as one is "iterating" (parameterized)

[1] A module $M : \mathbb{X} \longrightarrow \mathbb{Y}$ is variously called a profunctor, a distributor, a *bi*module: it is equivalently a functor $M : \mathbb{X}^{\mathrm{op}} \times \mathbb{Y} \longrightarrow \mathbf{Set}$ or a "bipartite" category consisting of the categories $\mathbb{X}$ and $\mathbb{Y}$ and in addition "cross maps" running from the objects of $\mathbb{X}$ to objects of $\mathbb{Y}$ – but not in the reverse direction.

constant time computations, that one keeps within polynomial time.

Of course, a categorical doctrine which demands that the player world should be affine does not exclude, as a model, a system which has duplication as well. Thus, the systems above are not ruled out. However, Hofmann's reason for insisting on an affine world was more fundamental: he had observed that one could not allow certain reasonable patterns of recursion over trees – such as counting their leaves (see section 3.6 and 5.5) – at the same time as allowing their duplication. This is because, with duplication, one can easily construct an exponential size tree by simply repeatedly adding a root node whose children are duplicates of the tree constructed so far. Counting the leaves of such a tree, of course, then takes one outside PTIME. Leivant's system evaded this problem by favoring a recursion principle which did not permit one to count the leaves of a tree while Bellantoni and Cook's system simple did not consider trees.

There is, thus, a trade-off between the power of the recursion principle and allowing duplication in the player world. In the systems presented here, the player world is essentially assumed to be higher-order (affine closed) and this commits us to a powerful recursion principle and, whence, to limiting duplication. Of course, we could have followed the approach of allowing duplication and limiting the recursion principle. However, this approach is at considerable cost to expressiveness: and this did not seem to serve the objective of providing a usable polynomial programming system.

The pursuit of expressive power, in fact, led us to deploy a programming construct called a "peek". This allows one to inspect data without actually using it. In the affine world, this really allows one to have one's cake and eat it and this – not surprisingly – enhances expressive power. Indeed, the construct leads one to the boundary between PTIME and PSPACE and requires, in order to fully express the type semantics, the introduction of lazy products into the p-world which distribute over coproducts. On the type theoretic side, the introduction of these lazy products and the necessity to express this distributive law requires a more sophisticated "bunched" system. The resulting type system, with an unconstrained distributive law, is too expresive as it produces a PSPACE sound and complete system. In order to recapture PTIME soundness it is necessary to cut the system down (see section 6.3). Not surprisingly, perhaps, this is achieved by controlling the use of "higher-order" types[2] in the bunched contexts. Categorically this corresponds precisely to disallowing the distribution (with respect to the lazy product) of "higher-order" types over coproducts.

The polarized type systems, introduced in this paper, correspond to the categorical doctrines and are introduced primarily to support the detailed bound calculations required to establish soundness of the initial models with respect to the various complexity classes. They are also the type systems which underly the programming language Pola [4]. The bounding arguments rely directly on the operational semantics which are introduced for evaluating programs (closed terms). Thus, the arguments are ultimately about the number of steps involved in an evaluation and the size of the structures required to support this evaluation.

It is reasonably to wonder whether we could not have relied on prior results, such as Cobham's characterization of PTIME or, indeed, on the existing systems for safe recursion, to supply a more economical proof. We claim this really is not an option because the systems are really quite different. It is interesting, for example, to consider what is involved in translating Cobham's characterization into the current system: rather disappointingly it appears that one is forced into a completely uninformative machine level description and the same thing happens if one tries to

[2]In fact, these include higher order, coinductive, *and* universl types, see section 6.3.

translate, say, Bellantoni and Cook's system. The source of difficulty is that both of these characterizations rely implicitly on duplication: however, showing that duplications can be implemented seems irretrievably to involve a low level argument which does not use to advantage any of the more powerful features of our system.

The fact that the setting introduced here supports coinductive data distinguishes it quite sharply from traditional setting for PTIME. In particular, the presence of coinductive data introduces structures which are "lazily" evaluated (see particularly the Leivant trees $\mathsf{LTree}(\mathsf{Bool})$ in section 6.1). This means that one can "hide" computations with behaviors which cannot be predicted by an evaluator. These hidden computations intuitively introduce a game or communication aspect into the evaluation which allows one to show that, in this setting (given a reasonable interpretation of these notions), P $\neq$ NP because these types allow easy access to distinguishing computations. These complexity aspects of the setting are briefly touched on and clearly deserve further attention.

2 The basic categorical setting

The basic categorical framework of this paper consists of a cartesian category $\mathbb{X}$, a category $\mathbb{Y}$, and a module connecting $\mathbb{X} \times \mathbb{Y} \longrightarrow \mathbb{Y}$ creating a polarized category [6] which is, in addition *strong*. The category $\mathbb{X}$ will referred to as the **opponent category** (or opponent world) and the category $\mathbb{Y}$ is referred to as the **player category** (or player world) while the module maps are referred to as **cross maps**.

Polarized strong categories are closely related to certain fibrations (over the opponent world). This provides an alternative and compelling perspective on these setting which we shall, in particular, exploit to provide the corresponding type theory. This section, therefore, develops the relationship to fibrations and also introduces a running example, $\mathcal{R}$-sized sets, which illustrates rather concletely the idea of the theory. Ultimately, we wish to examine the initial models of the doctrines, however, to do this effectively requires the type theoretic machinery of the later sections 4 and 5.

2.1 Polarized strong categories

A **polarized strong category** consists of a cartesian category[3] $\mathbb{X}$ (the opponent world), and a category $\mathbb{Y}$ (the player world), and a module M:

$$M : \mathbb{X} \times \mathbb{Y} \longrightarrow \mathbb{Y}$$

equipped with a "strong" composition and "strong" identities for the module maps:

$$\frac{(X_1, Y_1) \xrightarrow{f} Y_2 \quad (X_2, Y_2) \xrightarrow{g} Y_3}{(X_1 \times X_2, Y_1) \xrightarrow{f;g} Y_3} \qquad \frac{}{(\mathbf{1}, Y) \xrightarrow{i_Y} Y}$$

which satisfy:

- Strong identities are natural: $i_Y y = (1, y) i_{Y'}$, for any $y : Y \longrightarrow Y'$ in $\mathbb{Y}$;
- The strong composition preserves the basic module structure: $(f; f')y = f; (f'y)$, $(x_1 \times x_2, y)(f_1; f_2) = ((x_1, y)f_1); (x_2, 1)f_2$, and $(1 \times x, 1)((fy); f') = f; (x, y)f'$;
- $(\pi_1, 1)f = i_Y; f : (\mathbf{1} \times X, Y) \longrightarrow Y'$ and $(\pi_0, 1)f = f; i_{Y'} : (X \times \mathbf{1}, Y) \longrightarrow Y'$;
- $(a_\times, 1)((f_1; f_2); f_3) = f_1; (f_2; f_3) : (X_1 \times (X_2 \times X_3), Y) \longrightarrow Y'$.

In order to demonstate the connection to fibrations we shall want to consider a fixed opponent world and a varying player world and to facilitate this we shall refer to a polarized strong category with opponent world $\mathbb{X}$ as an **$\mathbb{X}$-strong category**. An **$\mathbb{X}$-strong functor** $F : \mathbb{Y} \longrightarrow \mathbb{Y}'$ between $\mathbb{X}$-strong categories will then be an ordinary functor $F : \mathbb{Y} \longrightarrow \mathbb{Y}'$ and a morphism, also labeled F, on cross maps such that:

$$\frac{(X, Y) \xrightarrow{f} Y'}{(X, F(Y)) \xrightarrow{F(f)} F(Y')}$$

[3]Here this means having finite products. In fact, having a tensor suffices for the basic strong composition structure; see [25].

which preserve the basic module structure $(x, F(y))F(h)F(y') = F((x, y)hy')$ and preserves the strong composition and identities:

- $F(i_Y) = i_{F(Y)}$
- $F(f_1; f_2) = F(f_1); F(f_2)$

Clearly the composite of two $\mathbb{X}$-strong functors is again an $\mathbb{X}$-strong functor. An **$\mathbb{X}$-strong transformation** between strong functors is an ordinary transformation between the ordinary functors $\alpha : F \longrightarrow F'$ such that for cross maps h we have $(1, \alpha)F'(h) = F(h)\alpha$. We now have:

Proposition 2.1 *$\mathbb{X}$-strong categories, functors, and transformations form a 2-category, written* $\mathsf{Str}(\mathbb{X})$*.*

2.2 Fibrational interpretation

Given an $\mathbb{X}$-strong category $\mathbb{Y}$ a fibration $p : \widetilde{\mathbb{Y}} \longrightarrow \mathbb{X}$ can be constructed[4], called the **bundle fibration** $\mathsf{bun}(\mathbb{Y})$, where the total category $\widetilde{\mathbb{Y}}$ of this fibration is defined as follows:

Objects: pairs of objects $(X, Y) \in \mathbb{X} \times \mathbb{Y}$;

Maps: A map from (X_1, Y_1) to (X_2, Y_2) is a pair (x, h), where $x : X_1 \longrightarrow X_2$ in $\mathbb{X}$ and $h : (X_1, Y_1) \longrightarrow Y_2$ is a module map;

Composition: let $(x, h) : (X_1, Y_1) \longrightarrow (X_2, Y_2)$ and $(x', h') : (X_2, Y_2) \longrightarrow (X_3, Y_3)$; then composition is defined by $(xx', (\Delta, 1)(h; (x, 1)h')) : (X_1, Y_1) \longrightarrow (X_3, Y_3)$;

Identities: $(1_X, (!_X, 1)i_Y) : (X, Y) \to (X, Y)$.

It is not hard to check that $\widetilde{\mathbb{Y}}$ is a category and moreover gives rise to a fibration:

Proposition 2.2 *The* $\mathsf{bun}(\mathbb{Y})$ *given by the projection functor $p : \widetilde{\mathbb{Y}} \longrightarrow \mathbb{X}$ defined by $p(X, Y) = X$ and $p(x, h) = x$ is a fibration over $\mathbb{X}$ with a cleavage.*

Proof: For each map $x : X \longrightarrow X'$ and object (X', Y') over X', the cartesian lifting is defined as $f = (x, (!_X, 1)i_{Y'}) : (X, Y') \longrightarrow (X', Y')$. Indeed, f is cartesian over x as $p(f) = x$ and given $g = (z, h) : (Z_1, Z_2) \longrightarrow (X', Y')$ such that z factors as $x'x$, there exists a bundle map $m = (x', h) : (Z_1, Z_2) \longrightarrow (X, Y')$ such that $p(m) = x'$ and:

$$\begin{aligned} mf &= (x', h)(x, (!_X, 1)i_{Y'}) \\ &= (x'x, (\Delta, 1)(h; (x', 1)(!_X, 1)i_{Y'})) \\ &= (z, (\Delta, 1)(h; (!_{Z_1}, 1)i_{Y'})) \\ &= (z, (\Delta, 1)(1 \times !_{Z_1}, 1)(h; i_{Y'})) \\ &= (z, (\Delta, 1)(1 \times !_{Z_1}, 1)(\pi_0, 1)h) \\ &= (z, h) = g \end{aligned}$$

Moreover, if $m' = (x'', h')$ also satisfies these conditions, then $x'' = p(m') = x'$ and $m'f = (z, h') = g = (z, h)$, so $h = h'$ and m is unique. □

[4] This is not the usual Grothendieck fibration from the module but uses the extra composition ';' of an $\mathbb{X}$-strong category in an essential way.

Proposition 2.3 *This assignment extends to a 2-functor* $\mathsf{bun} : \mathsf{Str}(\mathbb{X}) \longrightarrow \mathsf{CFib}(\mathbb{X})$ *which, moreover, preserves products.*

PROOF: The preservation of products is immediate from the construction.

An $\mathbb{X}$-strong functor F extends to a cartesian functor $\widetilde{F}$ on the fibration as follows:

$$\begin{aligned}\widetilde{F}(X,Y) &= (X, F(Y)) \\ \widetilde{F}(x,h) &= (x, F(h))\end{aligned}$$

This preserves the cleavage as $\widetilde{F}(x,(!_X,1)i_Y) = (x, F((!_X,1)i_Y)) = (x,(!_X,1)i_{F(Y)})$ and is a functor as:

$$\begin{aligned}\widetilde{F}(1_X, i_Y) &= (1_X, F((!_X, 1_Y)i_Y)) \\ &= (1_X, (!_X, F(1_Y))F(i_Y)) \\ &= (1_X, (!_X, 1_{F(Y)})i_{F(Y)})\end{aligned}$$

and:

$$\begin{aligned}\widetilde{F}(x,h);\widetilde{F}(x',h') &= (x,F(h))(x',F(h')) \\ &= (xx', (\Delta,1)(F(h);(x,1)F(h'))) \\ &= (xx', (\Delta,1)(F(h);F((x,1)h'))) \\ &= (xx', (\Delta,1)(F(h;(x,1)h'))) \\ &= (xx', F((\Delta,1)(h;(x,1)h'))) \\ &= \widetilde{F}((x,h)(x',h'))\end{aligned}$$

The cartesian lifting of a map $x : X \longrightarrow X'$ in $\mathbb{X}$ is $(x,(!_X,1)i_Y) : (X,Y) \longrightarrow (X',Y)$. This map is preserved as $\widetilde{F}((x,(!_X,1_Y)i_Y)) = (x,F((!_X,1_Y)i_Y)) = (x,(!_X,1_{F(Y)})i_{F(Y)})$. Therefore $\widetilde{F}$ is a cartesian functor.

A $\mathbb{X}$-strong natural transformation $\alpha : F \longrightarrow G$ extends to a cartesian natural transformation $\widetilde{\alpha} : \widetilde{F} \longrightarrow \widetilde{G}$ as follows: the components of the natural transformation are $(1_X,(!_X,1)\alpha_Y) : (X,F(Y)) \longrightarrow (X,G(Y))$. It is easy to check that this in fact defines a cartesian natural transformation. □

Given any cartesian category $\mathbb{X}$ then $\mathbb{X}$, itself, can be regarded as a $\mathbb{X}$-strong category by letting the cross maps be:

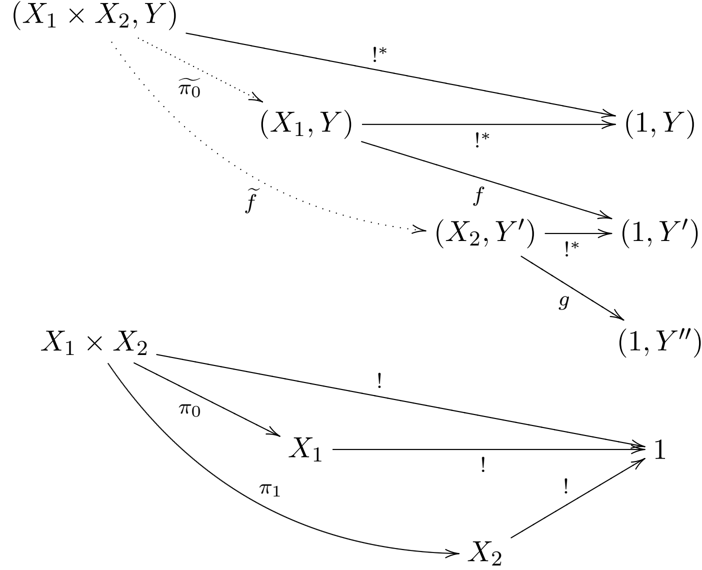
$$\frac{X_1 \times X \xrightarrow{h} X_2}{\overline{(X,X_1) \xrightarrow{h} X_2}}$$

This makes these cross maps "maps in context". Note that this makes $f;g = a_\times(f \times 1)g$. A $\mathbb{X}$-strong functor F then becomes a strong functor in the usual sense [15, 7, 25].

We now show that there is a functor in the reverse direction: that is, given any fibration (here we consider fibrations with a cleavage) over a cartesian category $\mathbb{X}$ the fiber over $\mathbf{1}$ naturally forms an $\mathbb{X}$-strong category.

Proposition 2.4 *There is a 2-functor* $\mathsf{pol} : \mathsf{CFib}(\mathbb{X}) \longrightarrow \mathsf{Str}(\mathbb{X})$.

Proof: (Sketch) Let $p : \mathbb{Y} \longrightarrow \mathbb{X}$ be a fibration with cleavage. Then we may build an $\mathbb{X}$-strong category on the fiber over $\mathbf{1}$, $p^{-1}(\mathbf{1})$, where one defines a cross map of the form $(X, Y) \longrightarrow Y'$ as a map $!_X^*(Y) \longrightarrow Y'$ in $\mathbb{Y}$. The strong identity maps are the identity maps in $p^{-1}(\mathbf{1})$. The strong composition of $(X_1, Y) = !_{X_1}^*(Y) \xrightarrow{f} Y'$ and $(X_2, Y') = !_{X_2}^*(Y') \xrightarrow{g} Y''$ is given by lifting the first map to the map $(X_1 \times X_2, Y) \xrightarrow{\widetilde{f}} (X_2, Y')$ (as illustrated below) and composing with g:



A morphism of fibrations $F : (p : \mathbb{Y} \longrightarrow \mathbb{X}) \longrightarrow (q : \mathbb{Y}' \longrightarrow \mathbb{X})$ consists of a functor $F : \mathbb{Y} \longrightarrow \mathbb{Y}'$ such that $p = F; q$ which preserves the cleavage. The restriction to the fiber over $\mathbf{1}$ then defines an $\mathbb{X}$ strong functor between the induced $\mathbb{X}$-strong categories. Similarly a natural transformation between a morphism of fibrations induces an $\mathbb{X}$-strong transformation. $\square$

Proposition 2.5 *The above functors forms a Galois adjunction*[5] $\mathsf{bun} \dashv \mathsf{pol} : \mathsf{Str}(\mathbb{X}) \longrightarrow \mathsf{CFib}(\mathbb{X})$.

Proof: (Sketch) The unit of the adjunction carries $\mathbb{Y}$ to the fiber over $\mathbf{1}$ in $\mathsf{bun}(\mathbb{Y})$:

$$\eta : \mathbb{Y} \longrightarrow \mathsf{pol}(\mathsf{bun}(\mathbb{Y})); \quad \begin{array}{c} Y \\ \downarrow f \\ Y' \end{array} \quad \mapsto \quad \begin{array}{c} (\mathbf{1}, Y) \\ \downarrow (1_{\mathbf{1}}, i_Y f) \\ (\mathbf{1}, Y') \end{array}$$

We need to show the following universal property:

$$\begin{array}{ccc} \mathbb{Y} & \xrightarrow{\eta} & \mathsf{pol}(\mathsf{bun}(\mathbb{Y})) \\ & \searrow^{H} & \vdots\ \mathsf{pol}(H^\sharp) \\ & & \mathsf{pol}(\mathbb{A}) \end{array}$$

[5] This is an adjunction $(\eta, \epsilon) : F \dashv G : \mathbb{X} \longrightarrow \mathbb{Y}$ with the additional property that $F(\eta)$ is an isomorphism – or equivalently $G(\epsilon)$ is an isomorphism. Any such adjunction can be factorized as a reflection followed by a coreflection.

To do this we indicate how $H^\sharp$ is defined on $(x, f) \in \mathsf{bun}(\mathbb{Y})$ using the lifting property (dotted maps) of the fibration $\mathbb{A}$ and the definition $(X, A) :=!_X^*(Y) \in p^{-1}(X)$ as above:

$$\begin{array}{ccccc}
(X,Y) & \overset{\widetilde{H(f)}}{\cdots\cdots>} & (X,Y') & \overset{\widetilde{(1,x)}}{\cdots>} & (X',Y') \\
& {\scriptstyle H(f)}\searrow & \swarrow{\scriptstyle !^*} & & \\
& & (1,Y') & \xleftarrow{\;!^*\;} & (X',Y')
\end{array}
\qquad H^\sharp(x,f) : (X,Y) \cdots> (X',Y')$$

$$\begin{array}{ccccc}
X & \xrightarrow{1} & X & \xrightarrow{x} & X' \\
& {\scriptstyle !}\searrow & \swarrow{\scriptstyle !} & & \\
& & 1 & \xleftarrow{\;!\;} & X'
\end{array}
\qquad x : X \longrightarrow X'$$

This is clearly unique as the liftings are unique.

To show this is a Galois adjunction it suffices to check that $\mathsf{bun}(\eta)$ is an isomorphism: however, this morphism of fibration is determined by its effect on the fiber over $\mathbf{1}$ and these fibers are essentially the same category. $\Box$

The fact that there is a Galois adjunction between $\mathsf{Str}(\mathbb{X})$ and $\mathsf{CFib}(\mathbb{X})$ means that one can identify a common full subcategory: the subcatgory of $\mathsf{CFib}(\mathbb{X})$ corresponds to "bundle fibrations" while the full subcategory of $\mathsf{Str}(\mathbb{X})$ corresponds more prosaically to the $\mathbb{X}$-strong categories in which $\mathbb{Y}$ is already the fibre over $\mathbf{1}$ in $\mathsf{bun}(\mathbb{Y})$.

2.3 The category of $\mathcal{R}$-sized-sets

This section introduces a concrete example which serves to illustrate the theory we are developing.

Let $\mathcal{R} = (\mathcal{R}, +, \cdot, 0, 1, \leq)$ be a size rig: that is an ordered (commutative) rig $\mathcal{R}$ with bottom element 0 (that is $0 \leq r$ for all $r \in \mathcal{R}$), and order-preserving operations. The canonical rig we have in mind is the natural numbers $\mathbb{N}$ (but $\mathbb{R}_{\geq 0}$ or $\mathbb{Q}_{\geq 0}$ will also do).

The category of $\mathcal{R}$-sized sets with polynomially bounded maps, denoted $\mathcal{R}\text{-}\mathsf{Set}_{\mathsf{poly}}$, is constructed as follows:

Objects: an object is a function $\alpha : A \longrightarrow \mathcal{R}$. We may think of this as a set A, each of whose elements is assigned an abstract size (i.e. an element of $\mathcal{R}$). These are called $\mathcal{R}$-sized sets;

Maps: A map from $\alpha : A \longrightarrow \mathcal{R}$ to $\beta : B \longrightarrow \mathcal{R}$ is a function $f : A \longrightarrow B$ such that there exists a polynomial $p \in \mathcal{R}[x]$ satisfying: for all $a \in A$, $\beta(f(a)) \leq p(\alpha(a))$. In this case we say that p is a bound for f;

Composition: The usual function composition. The composite is bounded by substitution of polynomials. Identities are the identity functions with identity polynomial bounds.

It is clear that this forms a category. It is furthermore a cartesian category. The product of $\mathcal{R}$-sized sets $\alpha : A \longrightarrow R$ and $\beta : B \longrightarrow R$ is the sized set

$$A \times B \longrightarrow \mathcal{R}; (a, b) \mapsto \alpha(a) + \beta(b).$$

The projection maps, π_0 and π_1, are bounded by identity polynomials. Given $f : C \longrightarrow A$ and $g : C \longrightarrow B$, the tuple map $\langle f, g\rangle$ is bounded by $p + q$, where p is a bound for f and q is a bound for g. The terminal object is the $\mathcal{R}$-sized set $\tau : \{*\} \longrightarrow \mathcal{R}; * \mapsto 0$.

The category of $\mathcal{R}$-sized sets and maps with a constant bound, denoted $\mathcal{R}\text{-}\mathsf{Set}_{\mathsf{const}}$, is the subcategory of $\mathcal{R}\text{-}\mathsf{Set}_{\mathsf{poly}}$ consisting of $\mathcal{R}$-sized sets and functions $f : A \longrightarrow B$ such that there exists a *constant* $c \in \mathcal{R}$ satisfying: for all $a \in A$, $\beta(f(a)) \leq \alpha(a) + c$.

Proposition 2.6 $\mathcal{R}\text{-}\mathsf{Set}_{\mathsf{const}}$ *is an* $\mathcal{R}\text{-}\mathsf{Set}_{\mathsf{poly}}$*-strong category.*

PROOF: It remains to describe the module structure. A cross map $(A, B) \longrightarrow C$ is a function $f : A\times B \longrightarrow C$ such that there exists a polynomial bound $p \in \mathcal{R}[x]$ satisfying, for all $(a, b) \in A\times B$, $\gamma(f(a, b)) \leq p(\alpha(a)) + \beta(b)$.

The strong composition is defined as follows: given $f : (A_1, B_1) \longrightarrow B_2$ and $g : (A_2, B_2) \longrightarrow C$ bounded by p and q, respectively, their composite is defined by $(f; g)((a_1, a_2), b_1) = g(a_2, f(a_1, b_1))$ and is bounded by $p + q$ as:

$$\begin{array}{rcl}\gamma((f;g)((a_1,a_2),b_1)) & \leq & \gamma(g(a_2, f(a_1,b_1))) \\ & \leq & q(\alpha_2(a_2)) + \beta_2(f(a_1,b_1)) \\ & \leq & q(\alpha_2(a_2)) + p(\alpha_1(a_1)) + \beta_1(b_1) \\ & \leq & (p+q)(\alpha(a_1,a_2)) + \beta_1(b_1)\end{array}$$

The identity cross maps $i_A : \mathbf{1} \times A \to A$ are given by second projection and are bounded by 0. □

2.4 Affine structure, products and coproducts in $\mathbb{X}$-strong categories

The $\mathbb{X}$-strong categories we are interested in here have much more structure: the player category is affine closed[6] with products and coproducts. This section describes how this structure is defined for $\mathbb{X}$-strong categories: in the next section the corresponding fibrational interpretation is given and this is followed by the corresponding structure in $\mathcal{R}$-sized sets.

Recall, as mentioned in the introduction, that if we had wished to capture Bellantoni and Cook's or Leivant's systems for PTIME we would have to forgo the closedness at this stage.

An $\mathbb{X}$-strong category $\mathbb{Y}$ is **affine closed** in case $\mathbb{Y} = (\mathbb{Y}, \otimes, \multimap, \mathbf{1})$ is affine closed and this structure extends to the module:

$$\frac{(X,Y_1) \xrightarrow{f} Y_1' \quad (X,Y_2) \xrightarrow{g} Y_2'}{(X, Y_1\otimes Y_2) \xrightarrow{f\otimes g} Y_1'\otimes Y_2'} \qquad \frac{(X, Z\otimes Y_1) \xrightarrow{f} Y_2}{\overline{(X,Z) \xrightarrow{\mathsf{cur}(f)} Y_1 \multimap Y_2}}$$

These must satisfy the equations:

- The tensor product is an $\mathbb{X}$-strong bifunctor: $(f\otimes g); (f'\otimes g') = f; f'\otimes g; g'$, $i_{Y_1}\otimes i_{Y_2} = i_{Y_1\otimes Y_2}$. The monoidal natural isomorphism $a_\otimes, u^L_\otimes, u^R_\otimes, c_\otimes$ are $\mathbb{X}$-strong natural transformations: e.g. $(1, a_\otimes)((f \otimes g) \otimes h) = (f \otimes (g \otimes h))a_\otimes$, etc;
- The tensor product must behave well with the module structure: $(x, y_1 \otimes y_2)(f \otimes g) = (x, y_1)f \otimes (x, y_1)g$ and $(f \otimes g)(y_1 \otimes y_2) = fy_1 \otimes gy_2$;

[6] Affine closed in the sense that it is a symmetric monoidal closed category in which the tensor unit is terminal.

- For the closed structure: $(\mathsf{cur}(f)\otimes(!_X,1)i_A)\mathsf{ev} = f$. We also assume $(x,1)\mathsf{cur}(f) = \mathsf{cur}((x,1)f)$. And $h;\mathsf{cur}(f) = \mathsf{cur}((h \otimes (!,1)i);f)$ and $\mathsf{cur}((1,\mathsf{ev})i_B) = i_{A \multimap B}$.

An $\mathbb{X}$-strong category $\mathbb{Y}$ is **cartesian** in case $\mathbb{Y} = (\mathbb{Y}, \times, \mathbf{1})$ is cartesian and the cartesian structure extends to cross maps as well:

$$\frac{(X,Y) \xrightarrow{f} Y_1 \quad (X,Y) \xrightarrow{g} Y_2}{\overline{(X,Y) \xrightarrow{\langle f,g\rangle} Y_1 \times Y_2}} \qquad \frac{}{\overline{(X,Y) \xrightarrow{!_{X,Y}} \mathbf{1}}}$$

These must satisfy the following equations:

- $\langle f,g\rangle\pi_0 = f$, $\langle f,g\rangle\pi_1 = g$, and $\langle h\pi_0, h\pi_1\rangle = h$.
- The terminal object satisfies: for any cross map $f : (X,Y) \to \mathbf{1}$, $f =!_{X,Y}$.

An $\mathbb{X}$-strong category $\mathbb{Y}$ has **coproducts** in case the categories $\mathbb{X}$ and $\mathbb{Y}$ have coproducts which are distributive with respect to the product in $\mathbb{X}$ and with respect to the tensor in $\mathbb{Y}$ (this latter is forced if $\mathbb{Y}$ is closed affine). This means there is a (strength) map $d : Z\times(Y_1+Y_2) \longrightarrow Z\times Y_1+Z\times Y_2$ which is inverse to the natural map in the reverse direction. In addition we require that the coproducts work across the module in two ways:

$$\frac{(X_1,Y) \xrightarrow{h_1} X' \quad (X_2,Y) \xrightarrow{h_2} Y'}{\overline{(X_1+X_2,Y) \xrightarrow[\langle h_1|h_2\rangle]{} Y'}} \qquad \frac{(X,Y_1) \xrightarrow{h_1} X' \quad (X,Y_2) \xrightarrow{h_2} Y'}{\overline{(X,Y_1+Y_2) \xrightarrow[\langle h_1|h_2\rangle]{} Y'}}$$

It is worth mentioning that we have not demanded that the products in $\mathbb{Y}$ distribute over the coproducts. This is quite deliberate as, although it is a very natural requirement, letting higher-order types distribute over products allows the expression of PSPACE complete problems. This will be is explained in section 5: however, it is based on an observation of Hofmann [13]. Notably this distributive law, is present in the example based on $\mathcal{R}$-sized sets – which should not be a surprise as they provide at least a PSPACE setting.

2.5 The fibrational interpretation

Products and coproducts can, of course, be defined at the 2-categorical level by demanding left and right adjoints to the diagonal $\mathbb{X}$-strong functor. The above presentation is a more explicit equivalent formulation. The functorial presentation, however, does have the advantage that it is transported along any 2-functor which preserves products: and bun is of this form. This leads to (part of) the following equivalent fibrational statement:

Proposition 2.7 *Let $\mathbb{Y}$ be an affine closed $\mathbb{X}$-strong category with products and coproducts. Then the corresponding fibration $p : \widetilde{\mathbb{Y}} \longrightarrow \mathbb{X}$ is fibered affine closed and has fibered products and coproducts. This means that each of the fibers is an affine closed category possessing products and coproducts, and this structure is preserved (on the nose) by the reindexing functors.*

Proof: (Sketch:) In the fiber over $X \in \mathbb{X}$ the products are given by:

$$(X,Y_1) \times (X,Y_2) = (X, Y_1 \times Y_2)$$

The terminal object in $p^{-1}(X)$ is $(X, \mathbf{1})$ and the unique map to it from any object (X, Y) is of just $(1_X, !_Y)$.

Similarly, if an $\mathbb{X}$-strong category $\mathbb{Y}$ has coproducts, then so do the fibers in the corresponding fibration:

$$(X, Y_1) + (X, Y_2) = (X, Y_1 + Y_2)$$

with injection $(1_X, (1, \sigma_0) i_{Y_1+Y_2})$ and $(1_X, (1, \sigma_1) i_{Y_1+Y_2})$. Given maps $(1_X, h_1) : (X, Y_1) \longrightarrow (X, Z)$ and $(1_X, h_2) : (X, Y_2) \longrightarrow (X, Z)$ the cotuple map is defined by $(1_X, \langle h_1 | h_2 \rangle) : (X, Y_1 + Y_2) \longrightarrow (X, Z)$.

If $\mathbb{Y}$ has distributive coproducts then then the isomorphism $d : Z \times (Y_1 + Y_2) \longrightarrow (Z \times Y_1) + (Z \times Y_2)$ lifts to the fiber $p^{-1}(X)$ as $(1_X, (!, d) i)$.

The affine closed structure lifts similarly:

$$\begin{array}{rcl} (X, Y_1) \otimes (X, Y_2) & = & (X, Y_1 \otimes Y_2) \\ (X, Y_1) \multimap (X, Y_2) & = & (X, Y_1 \multimap Y_2). \end{array}$$

Given a map $(1_X, h) : (X, Y_1 \otimes Z) \longrightarrow (X, Y_2)$ its curried form is the map $(1_X, \mathsf{cur}(h)) : (X, Z) \longrightarrow (X, Y_1 \multimap Y_2)$.

Finally, we note that the cartesian maps are of the form $(X_1, Y) \longrightarrow (X_2, Y)$ with the second component fixed, thus, the reindexing functors preserve all of this structure strictly. □

It follows that the total category $\widetilde{\mathbb{Y}}$ is itself cartesian (as the fibration is fibered cartesian and the base is cartesian). However, it does not, in general, inherit the coproduct structure.

2.6 The interpretation in $\mathcal{R}$-sized sets

In order to describe this additional structure in $\mathcal{R}$-sized sets it is necessary to assume that the size rig $\mathcal{R}$ has infima for all *nonempty* sets and that these are preserved by the operations. Note that this means that the maximum of any two elements in the rig can be defined by $\max(a, b) := \inf\{c \mid a, b \leq c\}$ as $a + b \in \{c \mid a, b \leq c\}$. Note that both $\mathbb{N}$ and $\mathbb{R}_{\geq 0}$ are examples of rigs with nonempty infima. With this additional assumption on $\mathcal{R}$, we have:

Proposition 2.8 $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$ *is affine closed.*

PROOF: The tensor structure on $\mathcal{R}$-sized sets is defined as follows. Given $\alpha : A \to \mathcal{R}$ and $\beta : B \to \mathcal{R}$ define

$$(\alpha \otimes \beta) : A \times B \longrightarrow \mathcal{R}; (a, b) \mapsto \alpha(a) + \beta(b).$$

The tensor product of maps $f : A_1 \to B_1$ and $g : A_2 \to B_2$ is defined by $(f \otimes g)(a_1, a_2) = (f(a_1), g(a_2))$ and is bounded by the sum of the constant bounds for f and g. The tensor is affine as the tensor unit is the same as the terminal object.

Given $\alpha : A \longrightarrow \mathcal{R}$ and $\beta : B \longrightarrow \mathcal{R}$, the internal hom is defined by $\alpha \multimap \beta : C(A, B) \longrightarrow \mathcal{R}$, where $C(A, B)$ is the set of constant $\mathcal{R}$-sized maps from A to B, and

$$(\alpha \multimap \beta)(f) = \inf \{c \mid \forall a \in A. \beta(f(a)) \leq \alpha(a) + c\}.$$

Given a map $f : A \times X \longrightarrow B$ of $\mathcal{R}$-sized sets bounded by $c \in \mathcal{R}$, define $\mathsf{cur}(f) : X \to C(A, B)$ by $x \mapsto \lambda a. f(a, x)$. This is bounded by the same constant $c \in \mathcal{R}$ as:

$$\begin{array}{rcl} (\alpha \multimap \beta)(\mathsf{cur}(f)(x)) & = & \inf \{c' \mid \forall a \in A. \beta(\mathsf{cur}(f)(x)(a)) \leq \alpha(a) + c'\} \\ & = & \inf \{c' \mid \forall a \in A. \beta(f(a, x)) \leq \alpha(a) + c'\} \\ & \leq & \xi(x) + c \end{array}$$

since $\beta(f(a,x)) \leq \alpha(a) + \xi(x) + c$. The evaluation map $\mathsf{ev} : (A \multimap B) \times A \longrightarrow B$ is given by $\mathsf{ev}(f, a) = f(a)$ and is bounded as there exists a constant $c \in \mathcal{R}$ such that:

$$\begin{aligned} \beta(f(a)) &\leq \alpha(a) + c \\ &\leq \alpha(a) + (\alpha \multimap \beta)(f) + c \end{aligned}$$

For each $\mathcal{R}$-sized set X define $C_X(A, B)$ to be the collection of functions $f : X \times A \longrightarrow B$ such that there exists $p \in \mathcal{R}[x]$ satisfying, for all $a \in A$ and $x \in X$, $\beta(f(x, a)) \leq p(\xi(x)) + \alpha(a)$. As before, arbitrary infima in $\mathcal{R}$ are required to assign a size to elements of $C_X(A, B)$ and to make this into an internal hom object. □

We now turn to the product structure for this example:

Proposition 2.9 *$\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$ is cartesian.*

Proof: Given $\mathcal{R}$-sized sets $\alpha : A \longrightarrow \mathcal{R}$ and $\beta : B \longrightarrow \mathcal{R}$, their cartesian product is given by the function

$$(\alpha \times \beta) : A \times B \longrightarrow \mathcal{R}; (a, b) \mapsto \max(\alpha(a), \beta(b)).$$

The tuple of maps $f : A \longrightarrow B$ and $g : A \longrightarrow C$, bounded by constants c_1 and c_2 respectively, is the map $\langle f, g\rangle : A \longrightarrow B \times C$, and is bounded by $\max(c_1, c_2)$. Projections are given by the usual projection functions and are bounded by 0.

This structure naturally extends to the cross maps as follows. Given $f : A \times B \rightarrow C_1$ and $g : A \times B \rightarrow C_2$, bounded by $p, q \in R[x]$, respectively, the pairing map $\langle f, g\rangle$ is bounded by $\max(p, q)$ as, for all a:

$$\begin{aligned} \gamma(\langle f, g\rangle(a, b)) &\leq \max(\gamma_1(f(a, b)), \gamma_2(g(a, b))) \\ &\leq \max(p(\alpha(a)) + \beta(b), q(\alpha(a)) + \beta(b)) \\ &\leq \max(p(\alpha(a)), q(\alpha(a))) + \beta(b) \end{aligned}$$

All of the equations are satisfied because they are satisfied in the underlying set-theoretic interpretation. □

Notice the diagonal map $d : A \longrightarrow A \otimes A$ is not in general bounded by a constant, so the tensor product and the cartesian product are not isomorphic in $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$. However, the two *are* isomorphic in the category $\mathcal{R}$-$\mathsf{Set}_{\mathsf{poly}}$, as the diagonal can be bounded by the polynomial $2x \in \mathcal{R}[x]$.

Proposition 2.10 *The polarized strong category $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$ has coproducts.*

Proof: Recall that this means that both the category $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$ and the category $\mathcal{R}$-$\mathsf{Set}_{\mathsf{poly}}$ have coproducts and that the coproduct acts on the module maps in two different ways. The coproduct of $\mathcal{R}$-sized sets $\alpha : A \rightarrow \mathcal{R}$ and $\beta : B \longrightarrow \mathcal{R}$ is defined as $\langle \alpha|\beta\rangle : A+B \rightarrow \mathcal{R}$, where $\langle \alpha|\beta\rangle(l, a) = \alpha(a)$ and $\langle \alpha|\beta\rangle(r, b) = \beta(b)$, where we have used '$l$' to denote the left component and 'r' to denote the right component of the disjoint union of A and B. The injections are the usual set-theoretic ones and are clearly bounded. Given $f : A \rightarrow C$ and $g : B \rightarrow C$, bounded by p and q, respectively, the copairing map $\langle f|g\rangle : A + B \rightarrow \mathcal{R}$ is bounded by $p + q$. Both categories have distributive coproducts as the map $d : A \times (B_1 + B_2) \rightarrow (A \times B_1) + (A \times B_2)$, defined by $d(a, (i, b)) = (i, (a, b))$,

is trivially bounded and is an isomorphism. Given cross maps $A \times B_1 \longrightarrow C$ and $A \times B_2 \longrightarrow C$, bounded by $p + c_1$ and $q + c_2$, respectively, the cotuple map $\langle f|g \rangle : A \times (B_1 + B_2) \longrightarrow C$ is defined by $\langle f|g \rangle (a, (l, b)) = f(a, b)$ and $\langle f|g \rangle (a, (r, b)) = g(a, b)$ and is again bounded by $p + q + \max(c_1, c_2)$. $\square$

3 Lifting, and comprehended recursion

An $\mathbb{X}$-strong category has a "lifting" if there is an $\mathbb{X}$-strong functor from the player category $\mathbb{Y}$ to the opponent category $\mathbb{X}$ which satisfies certain properties. Lifting plays an important role in the recursion principle described in the next section. It also has an appealing fibrational interpretation as it corresponds to the bundle fibration having "comprehension".

3.1 Lifting in $\mathbb{X}$-strong categories

We say that the module has a lift if for each $Y \in \mathbb{Y}$ there is an object $\uparrow(Y) \in \mathbb{X}$ and a module map

$$(\uparrow(Y), \mathbf{1}) \xrightarrow{\epsilon_Y} Y$$

such that whenever $(X, \mathbf{1}) \xrightarrow{h} Y$ is a module map there is a unique map $h^\flat : X \longrightarrow \uparrow(Y)$ making

$$\begin{array}{ccc} (X,\mathbf{1}) & \xrightarrow{h} & Y \\ {\scriptstyle (h^\flat,1)}\downarrow & \nearrow_{\epsilon_Y} & \\ (\uparrow(Y),\mathbf{1}) & & \end{array}$$

commute. The combinator $\flat$ is an operation which takes certain cross maps to $\mathbb{X}$-maps. We can define an operation in the other direction $\sharp$ by $g^\sharp = (g, 1)\epsilon_Y$. Then the following equations are easy consequences of the definition:

$$\begin{aligned} (x^\sharp)^\flat &= x \\ (h^\flat)^\sharp &= h \end{aligned}$$

We also have $((x, 1)hy)^\flat = xh^\flat(\epsilon y)^\flat$.

Proposition 3.1 *Lifting defines an $\mathbb{X}$-strong functor $\uparrow(-) : \mathbb{Y} \longrightarrow \mathbb{X}$ defined by $Y \mapsto \uparrow(Y)$ and $y : Y_1 \to Y_2 \mapsto (\epsilon_{Y_1} y)^\flat$.*

Proof: The identity is preserved as $\uparrow(1_Y) = (\epsilon_Y 1_Y)^\flat = (1^\sharp_{\uparrow(Y)})^\flat = 1_{\uparrow(Y)}$. Composition is preserved as:

$$\begin{aligned} \uparrow(y_1)\uparrow(y_2) &= (\epsilon_{Y_1} y_1)^\flat (\epsilon_{Y_2} y_2)^\flat \\ &= (((\epsilon_{Y_1} y_1)^\flat (\epsilon_{Y_2} y_2)^\flat)^\sharp)^\flat \\ &= (((\epsilon_{Y_1} y_1)^\flat (\epsilon_{Y_2} y_2)^\flat, 1)\epsilon_{Y_3})^\flat \\ &= (((\epsilon_{Y_1} y_1)^\flat, 1)((\epsilon_{Y_2} y_2)^\flat, 1)\epsilon_{Y_3})^\flat \\ &= (((\epsilon_{Y_1} y_1)^\flat, 1)\epsilon_{Y_2} y_2)^\flat \\ &= (\epsilon_{Y_1} y_1 y_2)^\flat \\ &= \uparrow(y_1 y_2) \end{aligned}$$

This shows that we have a mere functor from $\mathbb{Y}$ to $\mathbb{X}$. To show it is $\mathbb{X}$-strong we must define it on cross maps: Given a cross map $h : (X, Y_1) \to Y_2$ we define $(\epsilon_{Y_1}; h)^\flat : (\uparrow(Y_1) \times X) \longrightarrow \uparrow(Y_2)$. First we show that this behaves correctly with the module structure:

$$\begin{aligned} (\epsilon; (hy))^\flat &= ((\epsilon; h)y)^\flat \\ &= (\epsilon; h)^\flat (\epsilon y)^\flat \end{aligned}$$

and:

$$\begin{aligned}(\epsilon;(x,y)h)^\flat &= ((1\times x,1)(\epsilon y;h))^\flat\\ &= (1\times x)(\epsilon y;h)^\flat\\ &= (1\times x)(((\epsilon y)^\flat,1)\epsilon;h)^\flat\\ &= (1\times x)(((\epsilon y)^\flat\times 1,1)(\epsilon;h))^\flat\\ &= (1\times x)((\epsilon y)^\flat\times 1)(\epsilon;h)^\flat\\ &= ((\epsilon y)^\flat\times x)(\epsilon;h)^\flat\end{aligned}$$

Next we show that this preserves the identity and module composition. For the identity we have:

$$\begin{aligned}(\epsilon_Y;i_Y)^\flat &= ((\pi_0,1)\epsilon_Y)^\flat\\ &= \pi_0 : Y\times\mathbf{1}\longrightarrow Y\end{aligned}$$

which is the identity in $\mathbb{X}$ considered an $\mathbb{X}$-strong category. Next we check that it preserves the module composition:

$$\begin{aligned}(\epsilon_{Y_1};f)^\flat;(\epsilon_{Y_2};g)^\flat &= a_\times((\epsilon_{Y_1};f)^\flat\times 1)(\epsilon_{Y_2};g)^\flat\\ &= a_\times(((((\epsilon_{Y_1};f)^\flat\times 1),1)\epsilon_{Y_2};g)^\flat\\ &= a_\times(((((\epsilon_{Y_1};f)^\flat,1)\epsilon_{Y_2});g)^\flat\\ &= a_\times((\epsilon_{Y_1};f);g)^\flat\\ &= a_\times((a_\times^{-1},1)\epsilon_{Y_1};(f;g))^\flat\\ &= (\epsilon_{Y_1};(f;g))^\flat\end{aligned}$$

$\Box$

This allows one to define the **lift combinator**:

$$\frac{(X,Y\otimes Y')\xrightarrow{f} Y''}{(X\times\uparrow(Y),Y')\xrightarrow[f^\uparrow]{} Y''}$$

as follows:

$$\cfrac{\cfrac{\cfrac{(\uparrow(Y),\mathbf{1})\xrightarrow{\epsilon_Y} Y\quad(\uparrow(Y),Y')\xrightarrow{(!,1)i_{Y'}} Y'}{(\uparrow(Y),Y')\xrightarrow{\epsilon_{Y,Y'}} Y\otimes Y'}\qquad (X,Y\otimes Y')\xrightarrow{f} Y''}{(\uparrow(Y)\times X,Y')\xrightarrow{\epsilon_{Y,Y'};f} Y''}}{(X\times\uparrow(Y),Y')\xrightarrow[f^\uparrow]{} Y''}$$

with $\epsilon_{Y,Y'} = (1,u_L^{-1})(e_Y\otimes(!,1)i_{Y'})$ and $f^\uparrow = (c_\times,1)(\epsilon_{Y,Y'};f)$. Clearly, a cartesian category $\mathbb{X}$ regarded as a $\mathbb{X}$-strong category has a trivial lift given by the identity functor.

Lemma 3.2 *Lifting is iso-monoidal*[7] *for the products and tensor in $\mathbb{Y}$ onto the product is $\mathbb{X}$.*

[7] These are often called *strong* monoidal, however, the reader will appreciate that we have quite a few "strong" notions in this paper already!

PROOF: Because the product and tensor are affine we expect comonoidal maps $\uparrow(Y_1 \otimes Y_2) \longrightarrow \uparrow(Y_1) \times \uparrow(Y_2)$ and $\uparrow(Y_1 \times Y_2) \longrightarrow \uparrow(Y_1) \times \uparrow(Y_2)$. The maps in the reverse direction are given by:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{Y_1 \otimes Y_2 \longrightarrow Y_1 \otimes Y_2}}{(\mathbf{1}, Y_1 \otimes Y_2) \longrightarrow Y_1 \otimes Y_2}}{(\uparrow(Y_1), Y_2) \longrightarrow Y_1 \otimes Y_2}}{(\uparrow(Y_1) \times \uparrow(Y_2), \mathbf{1}) \longrightarrow Y_1 \otimes Y_2}}{\uparrow(Y_1) \times \uparrow(Y_2) \longrightarrow \uparrow(Y_1 \otimes Y_2)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{Y_1 \longrightarrow Y_1}}{(\mathbf{1}, Y_1) \longrightarrow Y_1}}{(\uparrow(Y_1), \mathbf{1}) \longrightarrow Y_1}}{(\uparrow(Y_1) \times \uparrow(Y_2), \mathbf{1}) \longrightarrow Y_1} \quad \dfrac{\dfrac{\dfrac{\overline{Y_2 \longrightarrow Y_2}}{(\mathbf{1}, Y_2) \longrightarrow Y_2}}{(\uparrow(Y_2), \mathbf{1}) \longrightarrow Y_2}}{(\uparrow(Y_1) \times \uparrow(Y_2), \mathbf{1}) \longrightarrow Y_2}}{(\uparrow(Y_1) \times \uparrow(Y_2), \mathbf{1}) \longrightarrow Y_1 \times Y_2}}{\uparrow(Y_1) \times \uparrow(Y_2) \longrightarrow \uparrow(Y_1 \times Y_2)}$$

Moreover, as $\mathbf{1}$ is terminal there is a unique map $!_{\uparrow(\mathbf{1})} : \uparrow(\mathbf{1}) \longrightarrow \mathbf{1}$ which is inverse to the map $(i_{\mathbf{1}})^\flat : \mathbf{1} \longrightarrow \uparrow(\mathbf{1})$. □

It is also important for our purpose that the lift preserves the coproduct structure as well and we shall simply *demand* that this is the case. That is, we shall demand that the canonical monoidal map is in fact an isomorphism.

3.2 Lifting in fibrations: comprehension

Recall that a fibration $p : \mathbb{Y} \to \mathbb{X}$, which has the terminal object functor $T : \mathbb{X} \longrightarrow \mathbb{Y}$, admits *comprehension* if this functor has a right adjoint [10].

Proposition 3.3 *If an $\mathbb{X}$-strong category $\mathbb{Y}$ has a lift, then the corresponding fibration $p : \widetilde{\mathbb{Y}} \longrightarrow \mathbb{X}$ admits comprehension in the above sense.*

PROOF: Let $\mathbb{Y}$ be an $\mathbb{X}$-strong category with a lift operator. Then there is a functor $R : \widetilde{\mathbb{Y}} \longrightarrow \mathbb{X}$ defined by $R(X,Y) = X \times \uparrow(Y)$ and $R(x,h) = \langle \pi_0 x, (h;\epsilon)^\flat \rangle$. We claim that this is right adjoint to the terminal object functor $T : \mathbb{X} \longrightarrow \widetilde{\mathbb{Y}}$, defined by $T(X) = (X, \mathbf{1})$ and $T(x) = (x, !)$. I.e. that there is a bijection:

$$\dfrac{(X, \mathbf{1}) \longrightarrow (X', Y)}{\overline{X \longrightarrow X' \times \uparrow(Y)}}$$

The unit and counit of the adjunction are:

$$\eta_X = \langle 1_X, !_X \rangle : X \to X \times \uparrow(\mathbf{1})$$

$$\epsilon_{X,Y} = (\pi_0, (\pi_1, 1)\epsilon_Y) : (X \times \uparrow(Y), \mathbf{1}) \longrightarrow (X, Y)$$

These are certainly natural transformations so it remains to verify the adjunction equations:

$$\begin{array}{rcl} T\eta; \epsilon T & = & (\langle 1_X, !_X \rangle, !); (\pi_0, (\pi_1, 1)\epsilon_{\mathbf{1}}) \\ & = & (1_X, (\Delta, 1)(!; (\langle 1_X, !_X \rangle, 1)(\pi_1, 1)\epsilon_{\mathbf{1}})) \\ & = & (1_X, !) \end{array}$$

and:

$$\begin{aligned}
\eta R R \epsilon &= \langle 1_{X\times\Uparrow(Y)}, !_{X\times\Uparrow(Y)}\rangle\langle \pi_0\pi_0, ((\pi_1,1)\epsilon_Y;\epsilon_{\mathbf{1}})^\flat\rangle \\
&= \langle 1_{X\times\Uparrow(Y)}, !_{X\times\Uparrow(Y)}\rangle\langle \pi_0\pi_0, ((\pi_1,1)\epsilon_Y;i_{\mathbf{1}})^\flat\rangle \\
&= \langle 1_{X\times\Uparrow(Y)}, !_{X\times\Uparrow(Y)}\rangle\langle \pi_0\pi_0, ((\pi_0,1)(\pi_1,1)\epsilon_Y)^\flat\rangle \\
&= \langle 1_{X\times\Uparrow(Y)}, !_{X\times\Uparrow(Y)}\rangle\langle \pi_0\pi_0, ((\pi_0\pi_1,1)\epsilon_Y)^\flat\rangle \\
&= \langle 1_{X\times\Uparrow(Y)}, !_{X\times\Uparrow(Y)}\rangle\langle \pi_0\pi_0, \pi_0\pi_1\rangle \\
&= \langle 1_{X\times\Uparrow(Y)}, !_{X\times\Uparrow(Y)}\rangle\pi_0 \\
&= 1_{X\times\Uparrow(Y)}
\end{aligned}$$

This completes the proof. □

3.3 Trivial lifts and lifting in $\mathcal{R}$-sized sets

It is convenient when we translate this categorical structure into type theory to view the lift as being the identity on types: we shall say that the lift is **trivial** in this case. Given a strong polarized category with a lift one can always extract (couniversally) from it a strong polarized category with a trivial lift. This shows that strong polarized categories with a trivial lift form a coreflexive subcategory of all strong polarized categories with a lift (and functors which preserve lift).

Proposition 3.4 *The full subcategory of strong polarized categories with a trivial lift is a coreflexive subcategory of the category of strong polarized categories with a lift.*

PROOF: For any strong polarized category $\mathbb{P}$ we must construct a strong polarized category with a trivial lift $T(\mathbb{P})$ and a functor $\epsilon : T(\mathbb{P}) \longrightarrow \mathbb{P}$ such that any functor from a strong polarized category with a trivial lift factorizes uniquely through this category. If $\mathbb{P} : \mathbb{X} \times \mathbb{Y} \longrightarrow \mathbb{X}$ then we set $T(\mathbb{P}) : \mathbb{Y} \times \mathbb{Y} \longrightarrow \mathbb{Y}$ where the o-category has homsets $T(\mathbb{P})_o(Y_1, Y_2) = \mathbb{X}(\uparrow(Y_1), \uparrow(Y_2))$ and the op-map $T_{op}((Y_0, Y_1), Y_2) = \mathbb{P}_{op}((\uparrow(Y_0), Y_1), Y_2)$ while the p-maps are the same: composition is then as in $\mathbb{P}$. The functor ϵ then carries o-objects their lifted counterpart.

That this now has the desired couniversal property is straightforward to check. □

Having a trivial lift greatly simplifies notation as lifting can be silent and implicitly managed by position. Throughout the development of the type theory we call assume a trivial lift.

The category of $\mathcal{R}$-sized sets has a trivial lift:

Proposition 3.5 *The $\mathcal{R}$-$\mathsf{Set}_{\mathsf{poly}}$-strong category $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$ has a trivial lift.*

PROOF: Lifting here is the identity on $\mathcal{R}$-sized sets and for any $\mathcal{R}$-sized set A there is a map $\epsilon_A : A \times \{*\} \longrightarrow A$ defined by $\epsilon_A(a, *) = a$. Then given any cross map $f : A \times \{*\} \longrightarrow B$, bounded by $p \in \mathcal{R}[x]$, there is a map $f^\flat : A \longrightarrow B$, uniquely defined by $f^\flat(a) = f(a, *)$, and is bounded by p as well. □

3.4 Polarized operators

Inductive data types in this setting correspond to "comprehended" initial fixed points for polarized operators. Polarized operators are more than $\mathbb{X}$-strong functors as they also act on cross maps:

$$\frac{(X,\mathbf{1}) \xrightarrow{h} Y'}{(F_o(X),\mathbf{1}) \xrightarrow[F_{op}(h)]{} F_p(Y')}\ F_{op}$$

A peculiarity of the (initial) fixed point calculus in his setting is that inductive data (i.e. fixed point data) does not in general supply material from which one can build more inductive data. This is because inductive data does not, in general, organize itself into a polarized operator.

The recursion principle which we present here is the "circular" principle due to Varmo Vene [24] and Luigi Santocanale [23]. It is based on using a "circular" combinator to determine maps from inductive types. From the programming perspective this is quite natural as the style is similar to a definition by recursion.

We show below that this circular recursion principle, when the player world is affine closed, is equivalent to a more usual looking initial algebra principle. Of course, one can also use the circular recursion principle when the player world is not affine closed: the result is a scheme which is strictly more expressive than the initial algebra scheme as it has some "built-in" higher-order.

We illustrate the recursion principle by using it to count the leaves of a (Hofmann) tree and to build an exponential (Leivant) tree. There are further, examples in the sections covering the type theory.

A **polarized operator** F on an $\mathbb{X}$-strong category consists of a pair of strong functors $F_p : \mathbb{Y}^n \longrightarrow \mathbb{Y}$, and $F_o : \mathbb{X}^n \longrightarrow \mathbb{X}$, and a map of cross maps F_{op}:

$$\frac{(X_1,\mathbf{1}) \xrightarrow{f_1} Y_1 \quad \cdots \quad (X_n,\mathbf{1}) \xrightarrow{f_n} Y_n}{(F_o(X_1,\ldots,X_n),\mathbf{1}) \xrightarrow{F_{op}(f_1,\ldots,f_n)} F_p(Y_1,\ldots,Y_n)}\ F_{op}$$

satisfying various natural conditions. In the unary case these are:

- F_o and F_p are $\mathbb{X}$-strong functors;
- $F_{op}((x,1)h) = (F_o(x),1)F_{op}(h)$ and $F_{op}(hy) = F_{op}(h)F_p(y)$.
- When there is a lift, lifting must preserve the polarized in the sense that there is a strong natural isomorphism γ^F such that:

$$\begin{array}{ccc} \mathbb{Y} & \xrightarrow{\uparrow} & \mathbb{X} \\ {\scriptstyle F_p}\downarrow & \Downarrow \gamma^F & \downarrow{\scriptstyle F_o} \\ \mathbb{Y} & \xrightarrow[\uparrow]{} & \mathbb{X} \end{array}$$

In settings with a trivial lift every object in the opponent world is actually a lifted player object. In such settings the polarized operators are completely determined by the player side. Thus, the specification of these operators in the player world is the determining aspect.

Below we list the polarized operators which are *always* present:

1. For any object A in $\mathbb{Y}$, the constant functors $K_p^A : \mathbb{Y}^0 \longrightarrow \mathbb{Y}$ and $K_o^A : \mathbb{X}^0 \longrightarrow \mathbb{X}$, defined by $K_p^A(Y) = A$ and $K_o^A(X) = \uparrow(A)$, form a polarized operator. In this case $K_{op}^A(*) = (!_{\uparrow(A)}, 1)\epsilon_A$.
2. The tensor product in $\mathbb{Y}$ and the product in $\mathbb{X}$ form a polarized operator.
3. The product in $\mathbb{Y}$ and the product in $\mathbb{X}$ form a polarized operator.
4. The coproduct in $\mathbb{Y}$ and coproduct in $\mathbb{X}$ form a polarized operator (this is why we required lifting to preserve coproducts).

Polarized operators compose as operations on a polarized strong category. Thus, further examples can be generated from the above basic examples. Thus, any "polynomial polarized operator", generated by $+, \times, \otimes$ and constants, is a polarized operator.

3.5 Inductive recursion principles

Let F be an n-ary polarized operator on a polarized strong category $\mathbb{Y}$. A **circular combinator**, $c[_]$ is a pair of assignments:

$$\frac{h : (X, Z \otimes D) \longrightarrow B}{c_p[h] : (X, F_p(Z) \otimes D) \longrightarrow B}$$

$$\frac{v : (X \times V, D) \longrightarrow B}{c_o[v] : (X \times F_o(V), D) \longrightarrow B}$$

in which X, B and D are fixed and called respectively the opponent context, the player context, and the base. This data is a *combinator* in the sense that it satisfies:

$$\begin{aligned} c_p[(1, f \otimes (!, 1)i_D)h] &= (1, F_p(f) \otimes (!, 1)i_D)c_p[h] \\ c_o[(\langle \pi_0, w \rangle, (!, 1)i_D)v] &= (\langle \pi_0, F_o(w) \rangle, (!, 1)i_D)c_o[v] \\ c_o[(\pi_0, r \otimes (!, 1)i_D)h] &= (\pi_0, F_{op}(r) \otimes (!, 1)i_D)c_p[h] \end{aligned}$$

Proposition 3.6 *In an affine closed polarized strong category circular combinators with contexts X and D and base B are in bijective correspondence to maps*

$$(X, F_p(D \multimap B)) \xrightarrow[g]{} D \multimap B.$$

PROOF: Given such a map g define the circular combinator by:

$$\cfrac{\cfrac{\cfrac{(X, Z \otimes D) \xrightarrow{f} B}{(X, Z) \xrightarrow{\mathsf{cur}(f)} D \multimap B}}{(X, F_p(Z)) \xrightarrow{F_p(\mathsf{cur}(f))} F_p(D \multimap B)} \quad (X, F_p(D \multimap B)) \xrightarrow{g} D \multimap B}{\cfrac{(X, F_p(Z)) \xrightarrow{F_p(\mathsf{cur}(f))g} D \multimap B}{(X, F_p(Z) \otimes D) \xrightarrow[c_p[f] = ((F_p(\mathsf{cur}(f))g) \otimes i_D)\mathsf{eval}]{} B}}$$

and

$$\cfrac{\cfrac{\cfrac{(X \times Z, D) \xrightarrow{f'} B}{(X \times Z, \mathbf{1}) \xrightarrow{\mathsf{cur}(f')} D \multimap B}}{(X \times F_o(Z), \mathbf{1}) \xrightarrow{\theta^F_X F_{op}(\mathsf{cur}(f'))} F_p(D \multimap B)} \quad (X, F_p(D \multimap B)) \xrightarrow{g} D \multimap B}{\cfrac{(X \times F_o(Z), \mathbf{1}) \xrightarrow{\theta^F_X F_{op}(\mathsf{cur}(f'))g} D \multimap B}{(X \times F_o(Z), D) \xrightarrow[c_o[f'] = ((\theta^F_X F_{op}(\mathsf{cur}(f'))g) \otimes i_D)\mathsf{eval}]{} B}}$$

Conversely take:

$$\cfrac{\cfrac{(X, (D \multimap B) \otimes D) \xrightarrow{\mathsf{ev}'} B}{(X, F_p(D \multimap B) \otimes D) \xrightarrow{c_p[\mathsf{ev}']} B}}{(X, F_p(D \multimap B)) \xrightarrow[\mathsf{cur}(c_p[\mathsf{ev}'])]{} D \multimap B}$$

□

There are two inductive (comprehended) recursion principles we wish to consider: the first is weaker than the second. The second incorporates the additional ability to use the data type itself within the recursion. This is a standard feature of primitive recursion as it is usually defined.

A **polarized inductive data type** in an affine polarized strong category for a polarized operator F is a fixed point $\mu y.F_p(y)$ of F_p, that is, there is an isomorphism:

$$\mathsf{cons} : F_p(\mu y.F_p(y)) \longrightarrow \mu y.F_p(y)$$

in $\mathbb{Y}$ satisfying a recursion principle. The **circular recursion principle** says that given any circular combinator $c[_]$ for F with contexts X and D over B then there is a unique map $\mu a.c[a] : (X \times \uparrow(\mu y.F_p(y)), D) \longrightarrow B$ making the following triangle commute:

$$\begin{array}{ccc} (X \times F_o(\uparrow(\mu y.F_p(y))), D) & \xrightarrow{(1 \times \Uparrow(\mathsf{cons}), 1)} & (X \times \uparrow(\mu y.F_p(y)), D) \\ & {\searrow}^{c_o[\mu a.c[a]]} \quad {\swarrow}_{\mu a.c[a]} & \\ & B & \end{array}$$

Notice that in the affine *closed* case we can resolve this to get a more usual looking fixed point property using the above proposition:

$$\begin{array}{ccc} (X \times F_o(\uparrow(\mu y.F_p(y))), \mathbf{1}) & \xrightarrow{(1 \times \Uparrow(\mathsf{cons}), 1)} & (X \times \uparrow(\mu y.F_p(y)), \mathbf{1}) \\ {\scriptstyle (\pi_1, F_{op}(\mathsf{cur}(\mu a.c[a])))} \downarrow & & \downarrow {\scriptstyle \mathsf{cur}(\mu a.c[a])} \\ (X, F_p(D \multimap B)) & \xrightarrow[\mathsf{cur}(c_p[(!,1)i\ \mathsf{ev}'])]{} & D \multimap B \end{array}$$

The **primitive recursion principle** says that given any circular combinator $c[_]$ for F with contexts X and D over B then there is a unique map $\mu^* a.c[a] : (X \times \uparrow(\mu^* y.F_p(y)), D) \longrightarrow B$ making

the following triangle commute:

$$\begin{array}{ccc} (X \times F_o(\uparrow(\mu y.F_p(y))), D) & \xrightarrow{(1\times\Uparrow(\mathsf{cons}),1)} & (X\times \uparrow(\mu y.F_p(y)), D) \\ \Big\downarrow{\scriptstyle (1\times\Delta,1)} & & \\ (X \times F_o(\uparrow(\mu y.F_p(y)))^2, D) & & \swarrow{\scriptstyle \mu^* a.c[a]} \\ & \searrow{\scriptstyle c_o[(1\times\pi_1)\mu^* a.c[a]]} \quad B & \end{array}$$

In the affine closed case, as above, we can resolve this to get a more usual looking primitive recursive principle:

$$\begin{array}{ccc} (X \times F_o(\uparrow(\mu y.F_p(y))), \mathbf{1}) & \xrightarrow{(1\times\Uparrow(\mathsf{cons}),1)} & (X\times \uparrow(\mu y.F_p(y)), \mathbf{1}) \\ \Big\downarrow{\scriptstyle (1,F_{op}(\mathsf{cur}(\mu^* a.c[a])))} & & \Big\downarrow{\scriptstyle \mathsf{cur}(\mu^* a.c[a])} \\ (X \times F_o(\uparrow(\mu y.F_p(y))), F_p(D \multimap B)) & \xrightarrow[\mathsf{cur}(c_p[(!,1)i\ \mathsf{ev}'])]{} & D \multimap B \end{array}$$

3.6 Counting the leaves of a tree

To illustrate how the circular recursion principle works consider the following example. Let $T'_p(A,B,C) = A + B \otimes C$ (and so $T'_o(X,Y,Z) = X + Y \times Z$) then define $\mathsf{Tree}_p(A) = \mu y.A + y \otimes y$. To remove unnecessary clutter we shall replace the type variable A by a constant. The fixed point isomorphism can be resolved into two constructors

$$\mathsf{Leaf} : A_p \longrightarrow \mathsf{Tree}_p \qquad \mathsf{node} : \mathsf{Tree}_p \otimes \mathsf{Tree}_p \longrightarrow \mathsf{Tree}_p.$$

We shall also need the (unary) natural numbers $\mathsf{Nat} = \mu y.\mathbf{1} + y$ with the constructors $\mathsf{zero} : \mathbf{1} \longrightarrow \mathsf{Nat}$ and $\mathsf{Succ} : \mathsf{Nat} \longrightarrow \mathsf{Nat}$ in order to count.

The circular combinator will have empty o-context and both the p-context and the base Nat. This means we must define a combinator:

$$\frac{(\mathbf{1}, X \otimes \mathsf{Nat}) \xrightarrow{f} \mathsf{Nat}}{(\mathbf{1}, (A + X \otimes X) \otimes \mathsf{Nat}) \xrightarrow[c[f]]{} \mathsf{Nat}}$$

we define $c[f]$ as:

$$(\mathbf{1}, (A + X \otimes X) \otimes \mathsf{Nat}) \xrightarrow{(1,\, i\ \mathsf{d}_\otimes)} (\mathbf{1}, A \otimes \mathsf{Nat} + X \otimes X \otimes \mathsf{Nat}) \xrightarrow{\langle i\ !\mathsf{Succ} | (1 \otimes f) f \rangle} \mathsf{Nat}$$

and then the map

$$(\uparrow(\mathsf{Tree}), \mathbf{1}) \xrightarrow{(1,\, (!,1)i\ \mathsf{Zero})} (\uparrow(\mathsf{Tree}), \mathsf{Nat}) \xrightarrow{\mu x.c[x]} \mathsf{Nat}$$

counts the leaves of the tree.

One can also build the exponential size Leivant style tree, however, we must use a slightly different datatype – which change is enough to remove the ability to count the leaves: $\mathsf{LTree} =$

$\mu y.A + y \times y$ which has constructors LLeaf and LNode. Note that nodes are now *products* of trees which one should regard as being lazy. For this tree we can construct the combinator:

$$\frac{(\mathbf{1}, X) \xrightarrow{h} \mathsf{LTree}}{(\mathbf{1}, 1 + X) \xrightarrow[d[h]]{} \mathsf{LTree}}$$

by defining $d[h]$ as $(\mathbf{1}, 1 + X) \xrightarrow{\langle i\ \mathsf{LLeaf} | f\ \Delta\ \mathsf{LNode} \rangle} \mathsf{LTree}$. This builds an exponential size tree (but based on products).

3.7 Circular recursion in $\mathcal{R}$-sized sets

$\mathcal{R}$-sized sets has inductive data types which satisfy the primitive recursive principle. However, it is somewhat easier to show that the weaker circular recursion principle holds. Here we limit ourselves to illustrating this point because when the type theory and its bounding arguments are developed in the next sections, it will become clear that the stronger primitive recursion principle also holds. It is interesting to note that the polynomial bounds for the circular recursion principle are particularly simple as they directly follow from the intuition due to Hofmannn discussed in the introduction.

Proposition 3.7 *The $\mathcal{R}$-$\mathsf{Set}_{\mathsf{poly}}$-strong category $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$ has all "polynomial" polarized inductive data types which satisfy the circular recursion principle.*

PROOF: Let $F(Z) = F_1(Z) + \cdots + F_k(Z)$ be a polarized operator generated by constants, coproducts, products and tensors in disjunctive normal form – which makes sense, in this setting, as both products and tensors distribute over coproducts. Such a functor always has a fixed point in Sets, F^*, which is the free algebra generated by the constructors cons_i. The size $\phi : F^* \longrightarrow \mathcal{R}$ is defined inductively by:

$$\begin{array}{rcl} \phi(\mathsf{cons}_i(d)) & = & 1 + \phi(d) \\ \phi(d_1; \ldots; d_n) & = & \max(\phi(d_1), \ldots, \phi(d_n)) \\ \phi(d_1, \ldots, d_n) & = & \sum_{l=1}^{n} \phi(d_l) \\ \phi(a) & = & \alpha(a) \quad (\text{where } \alpha : A \longrightarrow \mathcal{R} \text{ is parametric}) \end{array}$$

Each constructor increases the size of its input by 1, so they are certainly maps in the player category. The map cons above is the cotuple of these constructors, and so it too is bounded by a size increase of 1. The inverse map is non-size increasing and so is also in $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$. This shows that this object is a fixed point in $\mathcal{R}$-$\mathsf{Set}_{\mathsf{const}}$.

It is convenient, in order to bound the recursion principle to use the transformation of proposition 3.6 and derive the size bounds from the fixed point form of the map. This then has to be evaluated to obtain the map we actually want. It suffices to show we can polynomially bound the

map fold recursively defined by:

$$\begin{array}{ccc}
(X \times F_o(F^*), \mathbf{1}) & \xrightarrow{(1,(!,1)i\ \mathsf{cons}} & (X \times F^*, \mathbf{1}) \\
\downarrow{\scriptstyle \langle \pi_0, \theta^F_X \rangle, !)} & & \\
(X \times F_o(X \times F^*), \mathbf{1}) & & \downarrow{\scriptstyle \mathsf{fold}} \\
\downarrow{\scriptstyle (1, F_{op}(\mathsf{fold}))} & & \\
(X, F_p(D \multimap B)) & \xrightarrow[g]{} & D \multimap B
\end{array}$$

Clearly this depends on g which, being in the p-world, it has a size constant increase dependent only on the o-context, $P_g(\xi(x))$. Next consider the maps down the lefthand side: the first map is the strength and, depending on the form of F, will duplicate the context a number of times, as its effect is exactly the same as for F_p this is also a constant size increase bounded by a polynomial in the o-context, $P_\theta(\xi(x))$. As F is a polynomial functor the size of $F_{op}(\mathsf{fold})$ is bounded by a constant added to the size of its parameters: the only subtly being that the one parameter shown may actually occur many times and thus the bound can be written in the form:

$$\beta(F_{op}((x, \mathsf{fold}(x, z))) \leq k + \sum_{j=1,..,r} \beta(\mathsf{fold}(x, z_i))$$

where the z_j are the next largest recursive occurrences of the data type within z.

Thus there is a constant bound (in the context size) by which each constructor can increase the size and, as $\phi(x)$ always exceeds the number of constructors, we have:

$$\beta(\mathsf{fold}(x, z)) \leq \phi(z) \cdot (k + P_\theta(\xi(x)) + P_g(\xi(x))).$$

Giving a polynomial bound as required. □

3.8 Coinductive recursion principle

Polarized coinductive types are simpler as they are essentially coinductive data types for the player world in a given context for a player functor G_p. One can define their couniversal property using a cocircular combinator for G_p with context X and cobase A:

$$\frac{(X, A) \xrightarrow{g} B}{(X, A) \xrightarrow[\mathsf{d}[g]]{} G_p(B)}$$

which satifies the obvious coherence condition with respect to varying B in context.

Cocircular combinators are in bijective correspondence with coalgebras in context as the following proposition shows:

Proposition 3.8 *In an affine closed polarized strong category cocircular combinators with contexts X and cobase A are in bijective correspondence to maps*

$$(X, A) \xrightarrow[f]{} G_p(A)$$

PROOF: Given a map f define a co-circular combinatory by:

$$\frac{(X,A) \xrightarrow{f} G_p(A) \quad \dfrac{(X,A) \xrightarrow{g} B}{(X, G_p(A)) \xrightarrow{G_p(g)} G_p(B)}}{(X,A) \xrightarrow[\mathsf{d}[g] = (\Delta,1)(f; G_p(g))]{} G_p(B)}$$

Conversely, setting $g = (!, 1)i$ gives the generating map for the combinator as above:

$$(X,A) \xrightarrow{\mathsf{d}[(!,1)i_A]} G_p(A)$$

□

A **polarized coinductive data type** in an affine polarized strong category for a player functor G_p is a fixed point $\nu y.G_p(y)$ of G_p, that is, there is an isomorphism:

$$\mathsf{dest} : \mu z.G_p(z) \longrightarrow G_p(\nu z.G_p(z))$$

in $\mathbb{Y}$ satisfying the following couniversal property for coinductive types. More precisely, the **cocircular recursion principle** says that given a combinator d as above there is a unique map $\nu a.\mathsf{d}(a)$ such that:

$$\begin{array}{ccc} & (X,A) & \\ {}^{\nu a.\mathsf{d}[a]}\swarrow & & \searrow^{\mathsf{d}[\nu a.\mathsf{d}[a]]} \\ \nu z.G_p(z) & \xrightarrow[\mathsf{dest}]{} & G_p(\nu z.G_p(z)) \end{array}$$

commutes.

3.9 Infinite lists

To illustrate how the cocircular recursion principle works consider the following example. Let $G_p(A,X) = A \times X$ and define $\mathsf{Inf}(A) = \nu y.A \times y$. The fixed point isomorphism can be resolved into two destructors:

$$\mathsf{Head} : \mathsf{Inf}(A) \longrightarrow A \qquad \mathsf{Tail} : \mathsf{Inf}(A) \longrightarrow \mathsf{Inf}(A)$$

Note that a second kind of infinite list $\mathsf{Inf}'(A)$ can be defined as the fixed point of the player functor $G'_p(A,X) = A \otimes X$. It has only the one destructor:

$$\mathsf{Next} : \mathsf{Inf}'(A) \longrightarrow A \otimes \mathsf{Inf}'(A)$$

The example we wish to illustrate here uses the first kind of infinite list and the following combinator:

$$\frac{(\mathbf{1}, \mathsf{Nat}) \xrightarrow{g} \mathsf{Inf}(\mathsf{Nat})}{(\mathbf{1}, \mathsf{Nat}) \xrightarrow[\mathsf{d}[g]]{} \mathsf{Nat} \times \mathsf{Inf}(\mathsf{Nat})}$$

where $\mathsf{d}[g] = \langle \mathsf{Head}, (1, \mathsf{Succ})g) \rangle$. The unique map $\nu a.\mathsf{d}[a] : \mathsf{Nat} \longrightarrow \mathsf{Inf}(\mathsf{Nat})$, given by the couniversal property, represents the infinite list of natural numbers beginning at some fixed one. For example, the infinite list of natural numbers is defined by:

$$\mathsf{allnats} = \mathsf{Zero}; \nu a.\mathsf{d}[a] : \mathbf{1} \longrightarrow \mathsf{Inf}(\mathsf{Nat})$$

Indeed,

$$\begin{aligned}
\mathsf{allnats}; \mathsf{Tail}; \mathsf{Tail}; \mathsf{Head} &= (1, \mathsf{Zero})\nu a.\mathsf{d}[a]; \mathsf{Tail}; \mathsf{Tail}; \mathsf{Head} \\
&= \mathsf{Zero}; \mathsf{Succ}; \nu a.\mathsf{d}[a]; \mathsf{Tail}; \mathsf{Head} \\
&= \mathsf{Zero}; \mathsf{Succ}; \mathsf{Succ}; \nu a.\mathsf{d}[a]; \mathsf{Head} \\
&= \mathsf{Zero}; \mathsf{Succ}; \mathsf{Succ}
\end{aligned}$$

4 Simply typed Pola

In this section we introduce the simple type system for Pola which forms the basis for the more expressive systems introduced in later sections. In this simple type system every well-typed Pola program is polynomial time computable, and, conversely, it is possible to encode polynomial time Turing machines in this system. Thus, simply typed Pola is complete for polynomial time computations. However, as observed by Hofmann in [13], there are naturally formulated polynomial time functions which are very awkward to program in systems based on predicative recursion. This makes it desirable to enhance Pola's expressiveness by moving to the more expressive full system introduced in section 5.

We define a sequent calculus for Pola terms in a style amenable to type inference. Types are either atomic types $A, B, \ldots$, the tensor product of types $A \otimes B$, or data types delivered by inductive (and coinductive) data declarations. These last allow, for example, unary and binary numbers, lists, and trees. A sequent has the form:

$$\Gamma \mid \Sigma \vdash t : C$$

where Γ is referred to as the **opponent** context and Σ is referred to as the **player** context. A context is a mapping of variables to types and so order is not important and, as this is a mapping the variables must be distinct. The inference rules are presented in Figure 1.

Variables can be introduced either in the player or the opponent context by the identity rule. A variable in the player context can be lifted to the opponent side using the lift rule, but not vice versa. The cut rules are expressed by where val clauses. Notice that the player context must be empty in the left hypothesis of the opponent cut rule. The rules for typing a tuple (t_1, t_2) express the fact that ',' in the player context represents an affine tensor product, while ',' in the opponent context represents a cartesian product. Although we do not give the structural rules, weakening is admissible for both the player and opponent contexts. Contraction is admissible only in the opponent context and is implemented by:

$$\frac{\Gamma, x : X \mid \ \vdash x : X \quad \Gamma, x : X, x' : X \mid \Sigma \vdash t : Y}{\Gamma, x : X \mid \Sigma \vdash t \ \mathsf{where\ val}\ x' = x : Y}$$

The rule for eliminating a tuple is given by the pcase syntax. The keyword case is reserved for the corresponding construct in the opponent world.

4.1 Inductive data

Inductive data is added in Pola with a specification of the form:

$$\mathsf{data}\ \mathsf{A}(B) \to X = \{\mathsf{C}_i : F_i(X, B) \to X\}_{i=1,\ldots,k}$$

where $F = F_1 + \cdots + F_k$ is a polarized operator, and B is a list of parameter types. This style of presentation reflects the fact that these are initial (polarized) algebras. Using the polarized operators we know always to be present we define some Hofmann-style data types in Figure 2. The case and pcase rules express the fixed point properties of data types in both the opponent and player worlds and are used to define simple programs like the predecessor function on unary numbers and the head and tail programs on lists:

$$\frac{x : A \in \Gamma \text{ or } \Sigma}{\Gamma \mid \Sigma \vdash x : A}\ \text{id} \qquad \frac{\Gamma \mid \Sigma, x : A \vdash t : C}{\Gamma, x : A \mid \Sigma \vdash t : C}\ \text{lift}$$

$$\frac{\Gamma \mid \ \vdash s : A \quad \Gamma, x : A \mid \Sigma \vdash t : C}{\Gamma \mid \Sigma \vdash t \ \textsf{where val}\ \{x = s\} : C}\ \text{cut}_o \qquad \frac{\Gamma \mid \Delta \vdash s : A \quad \Gamma \mid \Sigma, x : A \vdash t : C}{\Gamma \mid \Sigma, \Delta \vdash t \ \textsf{where val}\ \{x := s\} : C}\ \text{cut}_p$$

$$\frac{\Gamma' \mid \Sigma' \vdash t' : C \qquad \begin{array}{c}[\Gamma' \mid \Sigma' \xrightarrow{f} C] \\ \vdots \\ \Gamma \mid \Sigma \vdash t : D\end{array}}{\Gamma \mid \Sigma \vdash t \ \textsf{where fun}\ \{f(\Gamma' \mid \Sigma') = t'\} : D}$$

$$\frac{}{\Gamma \mid \Sigma \vdash () : \mathbf{1}} \qquad \frac{\Gamma \mid \Delta \vdash s : \mathbf{1} \quad \Gamma \mid \Sigma \vdash t : Y}{\Gamma \mid \Sigma, \Delta \vdash \textsf{pcase}\ s\ \textsf{of}\ ().t : Y}$$

$$\frac{\Gamma \mid \Sigma_1 \vdash t_1 : A \quad \Gamma \mid \Sigma_2 \vdash t_1 : B}{\Gamma \mid \Sigma_1, \Sigma_2 \vdash (t_1, t_2) : A \otimes B} \qquad \frac{\Gamma \mid \Delta \vdash s : A \otimes B \quad \Gamma \mid \Sigma, x : A, y : B \vdash t : C}{\Gamma \mid \Sigma, \Delta \vdash \textsf{pcase}\ s\ \textsf{of}\ (x, y).t : C}$$

$$\frac{\Gamma \mid \Delta \vdash t : F_i(B, \mathsf{A}(B))}{\Gamma \mid \Delta \vdash \mathsf{C}_i(t) : \mathsf{A}(B)} \qquad \frac{\Gamma \mid \ \vdash s : \mathsf{A}(B) \quad \{\Gamma, x_i : F_i(B, \mathsf{A}(B)) \mid \Sigma \vdash t_i : C\}_{i=1}^{k}}{\Gamma \mid \Sigma \vdash \textsf{case}\ s\ \textsf{of}\ \{\mathsf{C}_i(x_i).t_i\} : C}$$

$$\frac{\Gamma \mid \Delta \vdash s : \mathsf{A}(B) \quad \{\Gamma \mid \Sigma, x_i : F_i(B, \mathsf{A}(B)) \vdash t_i : C\}_{i=1}^{k}}{\Gamma \mid \Sigma, \Delta \vdash \textsf{pcase}\ s\ \textsf{of}\ \{\mathsf{C}_i(x_i).t_i\} : C}$$

$$\frac{\begin{array}{|l}\forall X \qquad [\Gamma \mid X, E \xrightarrow{f} Y] \\ \left\{ \begin{array}{c} \vdots \\ \hline \Gamma, x_i' : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X, B), y : E \vdash t_i : Y \end{array} \right\}_{i=1}^{k}\end{array} \qquad \begin{array}{c}[\Gamma, \mathsf{A}(B) \mid E \xrightarrow{f} Y] \\ \vdots \\ \Gamma \mid \Delta \vdash s : Z\end{array}}{\Gamma \mid \Delta \vdash \textsf{fold}\ f(\#, y)\ \textsf{as}\ \{\mathsf{C}_i(x_i' \mid x_i).t_i\}\ \textsf{in}\ s : Z}\ \text{fold}$$

$$\frac{\{\Gamma' \mid \ \vdash s_i : X_i\}_{X_i \in \Gamma} \quad \{\Gamma' \mid \Sigma_i' \vdash t_i : E_i\}_{E_i \in \Sigma} \quad [\Gamma \mid \Sigma \xrightarrow{f} Y]}{\Gamma' \mid \Sigma_1', \ldots, \Sigma_r' \vdash f(s_1, \ldots, s_n \mid t_1, \ldots, t_r) : Y}\ \text{function use}$$

Figure 1: Simply typed Pola

$$\text{data Nat} \to X = \begin{cases} \text{Zero} : \mathbf{1} \to X \\ \text{Succ} : X \to X \end{cases} \qquad \text{data BNat} \to X = \begin{cases} \text{End} : \mathbf{1} \to X \\ \text{B0} : X \to X \\ \text{B1} : X \to X \end{cases}$$

$$\text{data Bool} \to X = \begin{cases} \text{True} : \mathbf{1} \to X \\ \text{False} : \mathbf{1} \to X \end{cases} \qquad \text{data L}(A) \to X = \begin{cases} \text{Nil} : \mathbf{1} \to X \\ \text{Cons} : A, X \to C \end{cases}$$

$$\text{data T}(A) \to X = \begin{cases} \text{Leaf} : A \to X \\ \text{Node} : X, X \to X \end{cases} \qquad \text{data F}_k \to X = \begin{cases} \text{S1} : \mathbf{1} \to X \\ \vdots \\ \text{Sk} : \mathbf{1} \to X \end{cases}$$

$$\text{data SF}(A) \to X = \begin{cases} \text{SS} : A \to X \\ \text{FF} : 1 \to X \end{cases} \qquad \text{data BN}(A) \to X = \begin{cases} \text{BZero} : A \to X \\ \text{BSucc} : X \to X \end{cases}$$

Figure 2: Inductive data types

$$\begin{array}{lll} \text{pred}(|x) & = & \text{pcase } x \text{ of} \\ & & \quad \text{Zero.Zero} \\ & & \quad \text{Succ}(n).n \end{array} \qquad \begin{array}{lll} \text{head}(|x) & = & \text{pcase } x \text{ of} \\ & & \quad \text{Nil.Nil} \\ & & \quad \text{Cons}(a, xs).a \end{array} \qquad \begin{array}{lll} \text{tail}(|x) & = & \text{pcase } x \text{ of} \\ & & \quad \text{Nil.Nil;} \\ & & \quad \text{Cons}(a, xs).xs \end{array}$$

To express all polynomial time functions we need to add a recursion principle, this is provided by the fold construct which implements a form of safe recursion. Its typing is given in Figure 1. The fold expression has the effect of storing a recursive function definition which may be accessed using the function call rule inside of the box. The proof inside the box must be parametric with respect to the universal type X; that is, the proof inside the box must be valid for *all* types X. Notice that once a universally typed variable is introduced, it must remain in the player context. This ensures that recursive calls are always placed in the affine player world and is the basic mechanism for ensuring polynomial time soundness. See Figure 3 for some examples which use the fold construct. Most of these are standard, except possibly the bnadd program. The Burroni natural numbers, $\mathsf{BN}(A)$, defined in Figure 2, can be used to represent the usual unary numbers of type Nat and even tensor products of unary numbers $\mathsf{Nat} \otimes \mathsf{Nat}$ can be represented by a single Burroni number of type $\mathsf{BN}(\mathsf{BN}(\mathbf{1}))$. The program bnadd is the unary addition function and has type $\mathsf{BN}(\mathsf{BN}(\mathbf{1}))\ | \quad \to \mathsf{BN}(\mathbf{1})$.

4.2 Operational semantics

We first define the class of **normal form elements** or **values** to be closed terms inductively defined by:

$$t ::= () \mid (t, t) \mid \mathsf{C}_i(t)$$

$$
\begin{array}{lcl}
\mathsf{add}(x|y) & = & \mathsf{fold}\ f(\#, z)\ \mathsf{as} \\
& & \quad \mathsf{Zero}.z \\
& & \quad \mathsf{Succ}(_|n).\mathsf{Succ}(f(n, z)) \\
& & \mathsf{in}\ f(x, y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{mul}(x, y|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
& & \quad \mathsf{Zero}.\mathsf{Zero} \\
& & \quad \mathsf{Succ}(_|n).\mathsf{add}(y|f(n)) \\
& & \mathsf{in}\ f(x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{parity}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
& & \quad \mathsf{Zero}.\mathsf{True} \\
& & \quad \mathsf{Succ}(_|n).\mathsf{pcase}\ f(n)\ \mathsf{of} \\
& & \quad\quad \mathsf{True}.\mathsf{False} \\
& & \quad\quad \mathsf{False}.\mathsf{True} \\
& & \mathsf{in}\ f(x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{tri}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
& & \quad \mathsf{Zero}.x \\
& & \quad \mathsf{Succ}(n'|n).\mathsf{add}(n'|f(n)) \\
& & \mathsf{in}\ f(x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{len}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
& & \quad \mathsf{Nil}.\mathsf{Zero} \\
& & \quad \mathsf{Cons}(_|a, _|w).\mathsf{Succ}(f(w)) \\
& & \mathsf{in}\ f(x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{rev}(x|) & = & \mathsf{fold}\ f(\#, y)\ \mathsf{as} \\
& & \quad \mathsf{Nil}.\mathsf{Nil} \\
& & \quad \mathsf{Cons}(_|a, _|w).f(w, \mathsf{Cons}(a, y)) \\
& & \mathsf{in}\ f(x, \mathsf{Nil})
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{sumtree}(x|) & = & \mathsf{fold}\ f(\#, y)\ \mathsf{as} \\
& & \quad \mathsf{Leaf}(n').\mathsf{add}(n'|y) \\
& & \quad \mathsf{Node}(_|x_1, _|x_2).f(x_1, f(x_2, y)) \\
& & \mathsf{in}\ f(x, \mathsf{Zero})
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{bnadd}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
& & \quad \mathsf{BZero}(n).n \\
& & \quad \mathsf{BSucc}(_|n).\mathsf{BSucc}(f(n)) \\
& & \mathsf{in}\ f(x)
\end{array}
$$

Figure 3: Example recursive Pola programs

These values or normal form elements form an $\mathbb{N}$-sized set as they can be inductively sized, $|_|$, as follows:

$$\begin{array}{rcl} |()| & = & 0 \\ |(t_1, t_2)| & = & |t_1| + |t_2| \\ |\mathsf{C}_i(t)| & = & 1 + |t| \end{array}$$

Closed terms are reduced to normal form according to the operational semantics (see Figure 4). We write $\Gamma \vdash t \Rightarrow e$ to mean that the term t reduces to the value e in the context Γ. The context Γ provides an assignment of values to variables in t. The context also stores the function environment, this consists of function definitions introduces by where fun and the recursive function definitions introduced by folds.

In general, we shall be interested in computing three bounds associated to any Pola term u:

α: A bound $\alpha[u]$ on the size of the normal form element which results from evaluating the term;

β: A bound $\beta[u]$ on the space required to evaluate the term;

γ: A bound $\gamma[u]$ on the time required to evaluate the term.

The first measure, α, is a polynomial bound – which depends on the sizes of the values used as arguments – on the size of the value produced by the evaluation. This does not say anything, of course, about the size of the intermediate values required in the computation or the time required for the computation.

The second measure β is more complicated as it is a measure the space required to do an evaluation. The state of an evaluation at any stage is given by a context, a stack of auxillary values (holding intermediate calculations) and (a pointer to) the current term which has to be evaluated. Thus the size is dominated by the sizes of the values in context and the stack of intermediate values: and this is what we shall take it to be.

It turns out that this calculation is not important at this stage as if one has a polynomial time bound one's storage is necessarily polynomially bounded. However, it shall important in the next section.

Finally, the way the time bound, γ, is measured is as the total size (number of nodes) of the proof tree for the evaluation. While most evaluation steps (nodes in the proof tree) can clearly be achieved in a constant time the steps that require one to retrieve the values of variables deserve some special comment.

In an implementation one can arrange that variables are essentially pointers so that indeed these retrievals are practically constant time. Theoretically, however, one may not be so comfortable with assuming this as it does depends on the model of computation. Thus, in particular, the model does affect what one means by "constant time": here we do have in mind a pointer model. However, in all the computations we shall consider the size of the contexts will be polynomially bounded so that overall, even if one assumes a linear retrieval time the computation will still be polynomial so that the main results are independent of the model of computation.

4.3 PTIME soundness

In this section we show that every well-typed Pola program is polynomial time computable. Our first objective is the size calculation:

$$\frac{}{x = e, \Gamma \vdash x \Rightarrow e}$$

$$\frac{\Gamma \vdash s \Rightarrow e' \quad x = e', \Gamma \vdash t \Rightarrow e}{\Gamma \vdash t \ \mathsf{where\ val}\ \{x = s\} \Rightarrow e}$$

$$\frac{f = \lambda xy.t', \Gamma \vdash t \Rightarrow e}{\Gamma \vdash t \ \mathsf{where\ fun}\ \{f(x,y) = t'\} \Rightarrow e}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow e_1 \quad \Gamma \vdash t_2 \Rightarrow e_2}{\Gamma \vdash (t_1, t_2) \Rightarrow (e_1, e_2)} \qquad \frac{\Gamma \vdash t \Rightarrow e}{\Gamma \vdash \mathsf{C}_i(t) \Rightarrow \mathsf{C}_i(e)}$$

$$\frac{\Gamma \vdash s \Rightarrow (e_1, e_2) \quad x_1 = e_1, x_2 = e_2, \Gamma \vdash t \Rightarrow e}{\Gamma \vdash \mathsf{pcase}\ s\ \mathsf{of}\ (x_1, x_2).t \Rightarrow e}$$

$$\frac{\Gamma \vdash s \Rightarrow \mathsf{C}_j(e') \quad x_j = e', \Gamma \vdash t_j \Rightarrow e}{\Gamma \vdash \mathsf{pcase}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\} \Rightarrow e}$$

$$\frac{\Gamma \vdash s \Rightarrow \mathsf{C}_j(e') \quad x_j = e', \Gamma \vdash t_j \Rightarrow e}{\Gamma \vdash \mathsf{case}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\} \Rightarrow e}$$

$$\frac{\Gamma \vdash s \Rightarrow \mathsf{C}_j(e') \quad x'_j = e', x_j = e', \Gamma \vdash t_j \Rightarrow e}{\Gamma \vdash \mathsf{pcase\ fold}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x'_i|x_i).t_i\} \Rightarrow e}$$

$$\frac{f = \lambda xy.\mathsf{pcase\ fold}\ x\ \mathsf{of}\ \{\mathsf{C}_i(x'_i|x_i).t_i\}, \Gamma \vdash s \Rightarrow e}{\Gamma \vdash \mathsf{fold}\ f(\#, y)\ \mathsf{as}\ \{\mathsf{C}_i(x'_i|x_i).t_i\}\ \mathsf{in}\ s \Rightarrow e}$$

$$\frac{\begin{array}{c} f = \cdots, \Gamma \vdash s_1 \Rightarrow e_1 \\ f = \cdots, \Gamma \vdash s_2 \Rightarrow e_2 \end{array} \quad x = e_1, y = e_2, f = \lambda xy.t, \Gamma \vdash t \Rightarrow e}{f = \lambda xy.t, \Gamma \vdash f(s_1, s_2) \Rightarrow e}$$

Figure 4: Operational semantics: inductive

Theorem 4.1 (size; inductive) *Suppose* $x_1 : X_1, \ldots, x_n : X_n \mid y_1 : Y_1, \ldots, y_m : Y_m \vdash u : Y$ *is provable in the simply typed* Pola*. Then there exists a polynomial* $\alpha[u](x_1, \ldots, x_n)$ *such that for all normal form inputs* $e_1 : X_1, \ldots, e_n : X_n$ *and* $a_1 : Y_1, \ldots, a_m : Y_m$*, whenever* $x_1 = e_1, \ldots, x_n = e_n, y_1 = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ *in the operational semantics,* $|e| \leq \alpha(|e_1|, \ldots, |e_n|) + \sum_{i=1}^{m} |a_i|$*.*

Notational conventions: when the normal form terms are clear from the context we shall write $s \Rightarrow e$ instead of $\Gamma \vdash s \Rightarrow e$ and $|e| \leq \alpha(\Gamma) + \Sigma$ instead of $|e| \leq \alpha[u](|e_1|, \ldots, |e_n|) + \sum_{i=1}^{m} |a_i|$.

Proof: We argue by structural induction on the term u.

case: $u \equiv x$ is a variable: If it is an opponent variable $\Gamma, x : X \mid \Sigma \vdash x : X$, then $\alpha[u](\Gamma, x) = x$; if it is a player variable $\Gamma \mid \Sigma, x : X \vdash x : X$, then $\alpha[u](\Gamma) = 0$.

case: $u \equiv t$ where $\{x := s\}$ and $\Gamma \mid \Sigma, \Delta \vdash u : C$. Then $\Gamma \mid \Delta \vdash s : A$ and $\Gamma \mid \Sigma, x : A \vdash t : C$. In this case we take $\alpha[u](\Gamma) = \alpha[s](\Gamma) + \alpha[t](\Gamma)$. For any normal form inputs to Γ, Δ, Σ, we have, by the induction hypothesis, $s \Rightarrow e'$ with $|e'| \leq \alpha[s](\Gamma) + \Delta$ and $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma) + |e'| + \Sigma$. Therefore $u \Rightarrow e$ and:

$$\begin{aligned} |e| &\leq \alpha[t](\Gamma) + |e'| + \Sigma \\ &\leq \alpha[t](\Gamma) + \alpha[s](\Gamma) + \Delta + \Sigma \end{aligned}$$

case: $u \equiv t$ where $\{x = s\}$ and $\Gamma \mid \Sigma \vdash u : C$. Then $\Gamma \mid \ \vdash s : A$ and $\Gamma, x : A \mid \Sigma \vdash t : C$. For any normal form inputs to Γ, Σ we have by the induction hypothesis $s \Rightarrow e'$ with $|e'| \leq \alpha[s](\Gamma)$ and $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma, |e'|) + \Sigma$. Therefore:

$$\begin{aligned} |e| &\leq \alpha[t](\Gamma, |e'|) + \Sigma \\ &\leq \alpha[t](\Gamma, \alpha[s](\Gamma)) + \Sigma \end{aligned}$$

case: $u \equiv (t_1, t_2)$ and $\Gamma \mid \Sigma \vdash u : X_1 \otimes X_2$, then $\Gamma \mid \Sigma_1 \vdash t_1 : X_1$ and $\Gamma \mid \Sigma_2 \vdash t_2 : X_2$, where $\Sigma = \Sigma_1, \Sigma_2$. For any normal form inputs to $\Gamma, \Sigma_1, \Sigma_2$ we have by the induction hypothesis $t_1 \Rightarrow e_1$ with $|e_1| \leq \alpha[t_1](\Gamma) + \Sigma_1$ and $t_2 \Rightarrow e_2$ with $|e_2| \leq \alpha[t_2](\Gamma) + \Sigma_2$. Therefore $(t_1, t_2) \Rightarrow (e_1, e_2)$ and:

$$\begin{aligned} |(e_1, e_2)| &= |e_1| + |e_2| \\ &\leq \alpha[t_1](\Gamma) + \alpha[t_2](\Gamma) + \Sigma_1 + \Sigma_2 \end{aligned}$$

case: $u \equiv$ pcase s of $(x_1, x_2).t$ and $\Gamma \mid \Sigma, \Delta \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : X_1 \otimes X_2$ and $\Gamma \mid \Sigma, x_1 : X_1, x_2 : X_2 \vdash t : Y$. For any normal form inputs to Γ, Σ, Δ we have by the induction hypothesis $s \Rightarrow (e_1, e_2)$ with $|(e_1, e_2)| = |e_1| + |e_2| \leq \alpha[s](\Gamma) + \Delta$, and $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma) + |e_1| + |e_2| + \Sigma$. Therefore $u \Rightarrow e$ and:

$$\begin{aligned} |e| &\leq \alpha[t](\Gamma) + |e_1| + |e_2| + \Sigma \\ &\leq \alpha[t](\Gamma) + \alpha[s](\Gamma) + \Sigma + \Delta \end{aligned}$$

case: $u \equiv \mathsf{C}_i(t)$ and $\Gamma \mid \Sigma \vdash u : \mathsf{A}(B)$. Then $\Gamma \mid \Sigma \vdash t : F_i(\mathsf{A}(B), B)$. For any normal form inputs to Γ, Σ we have by the induction hypothesis $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma) + \Sigma$. Therefore, $u \Rightarrow \mathsf{C}_i(e)$ and:

$$\begin{aligned} |\mathsf{C}_i(e)| &= |e| + 1 \\ &\leq \alpha[t](\Gamma) + \Sigma + 1 \end{aligned}$$

case: $u \equiv \mathsf{case}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma \vdash u : Y$. Then $\Gamma \mid\ \vdash s : \mathsf{A}(B)$ and $\Gamma, x_i : F_i(\mathsf{A}(B), B) \mid \Sigma \vdash t_i : Y$. For any normal form inputs to Γ, Σ we have by the induction hypothesis $s \Rightarrow \mathsf{C}_j(e')$ with $|\mathsf{C}_j(e')| = 1 + |e'| \leq \alpha[s](\Gamma)$, and $t_j \Rightarrow e$ with $|e| \leq \alpha[t_j](\Gamma, |e'|) + \Sigma$. Therefore, $u \Rightarrow e$ and:

$$\begin{aligned} |e| &\leq \alpha[t_j](\Gamma, |e'|)) + \Sigma \\ &\leq \max_i(\alpha[t_i](\Gamma, \alpha[s](\Gamma))) + \Sigma \end{aligned}$$

Note that s can evaluate to different constructors depending on the normal form inputs, so the max is used to give a uniform bound.

case: $u \equiv \mathsf{pcase}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma, \Delta \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : \mathsf{A}(B)$ and $\Gamma \mid \Sigma, x_i : F_i(\mathsf{A}(B), B) \vdash t_i : Y$. For any normal form inputs to Γ, Σ, Δ we have by the induction hypothesis $s \Rightarrow \mathsf{C}_j(e')$ with $|\mathsf{C}_j(e')| = 1 + |e'| \leq \alpha[s](\Gamma) + \Delta$, and $t_j \Rightarrow e$ with $|e| \leq \alpha[t_j](\Gamma) + |e'| + \Sigma$. Therefore, $u \Rightarrow e$ and:

$$\begin{aligned} |e| &\leq \alpha[t_j](\Gamma) + \Sigma + |e'| \\ &\leq \max_i(\alpha[t_i](\Gamma)) + \Sigma + \alpha[s](\Gamma) + \Delta \\ &\leq \max_i(\alpha[t_i](\Gamma)) + \alpha[s](\Gamma) + \Sigma + \Delta \end{aligned}$$

case: $u \equiv \mathsf{fold}\ f(\#, y)\ \mathsf{as}\ \{\mathsf{C}_i(x'_i|x_i).t_i\}\ \mathsf{in}\ s$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : Y$ and $\Gamma, x'_i : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X, B), y : E \vdash t_i : Y$. Then for any normal form inputs to Γ, Δ by the induction hypothesis we have $s \Rightarrow e$ with $|e| \leq \alpha[s](\Gamma) + \Delta$. Therefore, $u \Rightarrow e$ and $|e| \leq \alpha[s](\Gamma) + \Delta$.

case: $u \equiv f(s, t)$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid\ \vdash s : \mathsf{A}(B)$, $\Gamma \mid \Delta \vdash t : E$ and $\Gamma, x'_i : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X, B), y : E \vdash t_i : Y$. Also suppose $s \Rightarrow e'$ and $t \Rightarrow e''$. By the induction hypothesis, the size increase in one evaluation of the box is bounded by a polynomial $\max_i(\alpha[t_i](\Gamma, x'_i))$. By the typing restriction (i.e. the universal type is affine) the number of recursive calls is bounded by $|e'|$. Therefore, the normal form term e'' can increase by at most $|e'| \cdot \max_i(\alpha[t_i](\Gamma, |e'|))$. Therefore:

$$\begin{aligned} |e| &\leq |e'| \cdot \max_i(\alpha[t_i](\Gamma, |e'|)) + |e''| \\ &\leq \alpha[s](\Gamma) \cdot \max_i(\alpha[t_i](\Gamma, \alpha[s](\Gamma))) + \alpha[t](\Gamma) + \Delta \end{aligned}$$

This exhausts the cases and finishes the proof. □

A crucial step in this proof is the last step and the observation that the type system ensures that in a box the number of recursive calls to a function must be bounded by the size of the subject of the fold. Ensuring this is key to obtaining the size and time bound in this system.

Having this size bound we now show that every simply typed Pola program is polynomial time computable. This means, of course, that every Pola program is polynomial space bounded, so the β proof for the simple type system is unnecessary. The γ proof is as follows:

Theorem 4.2 (time; inductive) *Let $\Gamma \mid \Sigma \vdash u : Y$, where $\Gamma \equiv x_1 : X_1, \ldots, x_n : X_n$ and $\Sigma \equiv y_1 : Y_1, \ldots, y_m : Y_m$. Then there is a polynomial $\gamma[u](x_1, \ldots, x_n)$ with the property that for any normal form inputs, $e_1 : X_1, \ldots, e_n : X_n$ and $a_1 : Y_1, \ldots, y_m : Y_m$, the size of the evaluation tree of $x_i = e_i, y_i = a_i \vdash u \Rightarrow e$ is bounded by $\gamma[u](|e_1|, \ldots, |e_n|)$. In particular, the time bound does not depend on the size of the player context.*

Notational convention: when the normal form elements are clear from the context, we write $\|u\|_T$ to denote the size of the evaluation tree of $x_1 = e_1, \ldots, x_n = e_n, y_i = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ in the operational semantics.

Proof: The proof is by structural induction on the term u.

case: $u \equiv x$ is a variable. Then either it is typed as a player variable or as an opponent variable. In either case the evaluation tree is bounded by a constant.

case: $u \equiv t \text{ where } \{x := s\}$ and $\Gamma \mid \Sigma, \Delta \vdash u : C$. Then $\Gamma \mid \Delta \vdash s : A$ and $\Gamma \mid \Sigma, x : A \vdash t : C$. By the induction hypothesis applied to s and t, we have:

$$\begin{aligned} \|u\|_T &= \|s\|_T + \|t\|_T + 1 \\ &\leq \gamma[s](\Gamma) + \gamma[t](\Gamma) + 1 \end{aligned}$$

case: $u \equiv t \text{ where } \{x = s\}$ and $\Gamma \mid \Sigma \vdash u : C$. Then $\Gamma \mid \;\vdash s : A$ and $\Gamma, x : A \mid \Sigma \vdash t : C$. By the induction hypothesis applied to s and t, we have:

$$\begin{aligned} \|u\|_T &\leq \gamma[s](\Gamma) + \gamma[t](\Gamma, |e'|) + 1 \\ &\leq \gamma[s](\Gamma) + \gamma[t](\Gamma, \alpha[s](\Gamma)) + 1 \end{aligned}$$

where $s \Rightarrow e'$. Notice the use of the size bound α here.

case: $u \equiv (t_1, t_2)$ and $\Gamma \mid \Sigma \vdash u : X_1 \otimes X_2$, then $\Gamma \mid \Sigma_1 \vdash t_1 : X_1$ and $\Gamma \mid \Sigma_2 \vdash t_2 : X_2$, where $\Sigma = \Sigma_1, \Sigma_2$. By the induction hypothesis applied to t_1 and t_2, we have:

$$\begin{aligned} \|u\|_T &= \|t_1\|_T + \|t_2\|_T + 1 \\ &\leq \gamma[t_1](\Gamma) + \gamma[t_2](\Gamma) + 1 \end{aligned}$$

case: $u \equiv \text{pcase } s \text{ of } (x_1, x_2).t$ and $\Gamma \mid \Sigma, \Delta \vdash \text{pcase } s \text{ of } (x_1, x_2).t : Y$. Then $\Gamma \mid \Delta \vdash s : X_1 \otimes X_2$ and $\Gamma \mid \Sigma, x_1 : X_1, x_2 : X_2 \vdash t : Y$. By the induction hypothesis applied to s and t, we have:

$$\begin{aligned} \|u\|_T &= \|s\|_T + \|t\|_T + 1 \\ &\leq \gamma[s](\Gamma) + \gamma[t](\Gamma) + 1 \end{aligned}$$

case: $u \equiv \mathsf{C}_i(t)$ and $\Gamma \mid \Sigma \vdash u : \mathsf{A}(B)$. Then $\Gamma \mid \Sigma \vdash t : F_i(\mathsf{A}(B), B)$. By the induction hypothesis, we have:

$$\begin{array}{rcl} \|u\|_T & = & \|t\|_T + 1 \\ & \leq & \gamma[t](\Gamma) + 1 \end{array}$$

case: $u \equiv \mathsf{case}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma \vdash u : Y$. Then $\Gamma \mid\ \vdash s : \mathsf{A}(B)$ and $\Gamma, x_i : F_i(\mathsf{A}(B), B) \mid \Sigma \vdash t_i : Y$. By the induction hypothesis, we have:

$$\begin{array}{rcl} \|u\|_T & \leq & \gamma[s](\Gamma) + \max_i(\gamma[t_i](\Gamma, |e'|) + 1 \\ & \leq & \gamma[s](\Gamma) + \max_i(\gamma[t_i](\Gamma, \alpha[s](\Gamma))) + 1 \end{array}$$

where $s \Rightarrow e'$. Again, notice the use of the size bound α here.

case: $u \equiv \mathsf{pcase}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma, \Delta \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : \mathsf{A}(B)$ and $\Gamma \mid \Sigma, x_i : F_i(\mathsf{A}(B), B) \vdash t_i : Y$. By the induction hypothesis, we have:

$$\begin{array}{rcl} \|u\|_T & \leq & \|s\|_T + \max_i(\|t_i\|_T) + 1 \\ & \leq & \gamma[s](\Gamma) + \max_i(\gamma[t_i](\Gamma)) + 1 \end{array}$$

case: $u \equiv \mathsf{fold}\ f(\#, y)\ \mathsf{as}\ \{\mathsf{C}_i(x_i'|x_i).t_i\}\ \mathsf{in}\ s$ and $\Gamma \mid \Delta \vdash s : Y$. By the induction hypothesis, we have:

$$\begin{array}{rcl} \|u\|_T & \leq & \|s\|_T + 1 \\ & \leq & \gamma[s](\Gamma) + \Delta + 1 \end{array}$$

case: $u \equiv f(s, t)$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid\ \vdash s : \mathsf{A}(B)$, $\Gamma \mid \Delta \vdash t : E$ and $\Gamma, x_i' : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X, B), y : E \vdash t_i : Y$. By the induction hypothesis, we have time bounds $\gamma[t_i](\Gamma, x_i')$ for each of the t_i. By the affine typing of the universal type X the body of the fold can be evaluated at most $|e'|$ times, where $s \Rightarrow e'$. Therefore, by the induction hypothesis applied to s and t and the size bound $\alpha[s](\Gamma)$, we have:

$$\begin{array}{rcl} \|u\|_T & \leq & |e'| \cdot \max_i(\gamma[t_i](\Gamma, |e'|)) + \|s\|_T + \|t\|_T + 1 \\ & \leq & \alpha[s](\Gamma) \cdot \max_i(\gamma[t_i](\Gamma, \alpha[s](\Gamma))) + \gamma[s](\Gamma) + \gamma[t](\Gamma) + 1 \end{array}$$

This exhausts the cases and finishes the proof. □

4.4 PTIME completeness

Having established polynomial soundness for the simple type system, we now demonstrate that, in principle, any polynomial time algorithm can be expressed expressed in simply typed Pola. Hence Pola is complete for polynomial time computations.

A Turing machine with k states and 2 symbols can be represented by the type:

$$M \equiv \mathbf{F}_k \otimes \mathbf{F}_3 \otimes \mathsf{L}(\mathsf{Bool})^{\otimes 2}$$

where $\mathbf{F}_k$ is a finite type representing the states of the machine and $\mathbf{F}_3$ is a finite type representing the current symbol: either 0 or 1 or blank b. The tape is split into a left and right part and is represented by the two lists of booleans[8]. The transition function of the machine is encoded by a purely player program:

$$\mathsf{step} : M \to M$$

and there is a player program for initializing the machine $\mathsf{init} : \mathsf{L}(\mathsf{Bool}) \to M$ which writes the input on the right tape and puts the machine into the initial state, and a player program $\mathsf{out} : M \to \mathsf{L}(\mathsf{Bool})$ for returning the result of the computation. The program for iterating the transition function n times is given by:

$$\begin{aligned}\mathsf{iter}(n|m) \quad = \quad & \mathsf{fold}\ f(\#, m)\ \mathsf{as} \\ & \quad \mathsf{Zero}.m \\ & \quad \mathsf{Succ}(_|z).\mathsf{step}(f(z, m)) \\ & \mathsf{in}\ f(n, m)\end{aligned}$$

Now, suppose we have a Turing machine which runs in polynomial time $p(x)$ in the size of its input. As Pola can represent polynomials of arbitrary size, there is an opponent Pola program $\mathsf{P}(x)$ representing $p(x)$. The full computation of the machine is then represented by the program $\mathsf{out}(\mathsf{iter}(\mathsf{P}(\mathsf{len}(x))|\mathsf{init}(x))) : \mathsf{L}(\mathsf{Bool}) \mid \quad \to \mathsf{L}(\mathsf{Bool})$. It is in this sense that Pola is complete for polynomial time computations.

4.5 Coinductive types

One of the novel features of Pola is the ability to define coinductive types. At first glance it might seem surprising that one can define, for example, infinite lists in Pola and remain in polynomial time. But the polynomial bound remains intact because all coinductive terms in Pola are evaluated lazily. In the categorical semantics the player world is affine closed and has cartesian products. In Pola these are coinductive lazy constructs and are introduced in this section.

Coinductive data is defined with a specification of the form:

$$\mathsf{data}\ X \to \mathsf{A}(A, B) = \{\mathsf{D}_i : A_i, X \to G_i(X, B)\}_{i=1,\ldots,k}$$

where $G = G_1 \times \cdots \times G_k$ is a player functor, and A is a list of (contravariant) parameter types and B is a list of (covariant) parameter types. This formulation expresses the fact that these are (parameterized) final polarized algebras. Some simple examples of coinductive types are given in Figure 5. Notice, in particular, that cartesian products can be defined in Pola, and higher-order capabilities are added with the addition of linear function spaces. Also note that two sorts of infinite lists can be defined using the cartesian product in the first case and the tensor product in the second case. When a coinductive term is created it is wrapped up in a record waiting for destruction. The requirement that evaluation only proceed upon destruction is crucial for the polynomial time soundness of the system.

The fixed point rules for coinductive types and the unfold box rule are given Figure 6.

For example, the destruction rule together with the record rule (for $k = 1$) give the usual rules for linear function spaces. Indeed, we shall sometimes write $\lambda x.t$ as short for the record $(\mathsf{Eval} : x.t)$, though the former expression is not officially part of the syntax of the language.

[8] Note that two tape symbols suffices as the three symbols 0, 1 and b can be encoded, respectively, as, for example, 01,10, and 00, with the machine reading two symbols at a time.

$$\mathsf{data}\ X \to \mathsf{Inf}_1(B) = \begin{cases} \mathsf{Head} : X \to B \\ \mathsf{Tail} : X \to X \end{cases} \qquad \mathsf{data}\ X \to \mathsf{Inf}_2(B) = \begin{cases} \mathsf{Next} : X \to B \otimes X \end{cases}$$

$$\mathsf{data}\ X \to A \multimap B = \begin{cases} \mathsf{Eval} : A, X \to B \end{cases} \qquad \mathsf{data}\ X \to A \times B = \begin{cases} \mathsf{P0} : X \to A \\ \mathsf{P1} : X \to B \end{cases}$$

Figure 5: Coinductive data types

$$\frac{\Gamma \mid \Delta_1 \vdash s : A_i \quad \Gamma \mid \Delta_2 \vdash t : \mathsf{A}(A,B)}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathsf{D}_i(s,t) : G_i(\mathsf{A}(A,B), B)} \text{ dest.}$$

$$\frac{\{\Gamma \mid \Delta, a_i : A_i \vdash t_i : G_i(\mathsf{A}(A,B), B)\}_{i=1}^{k}}{\Gamma \mid \Delta \vdash (\mathsf{D}_i : a_i.t_i) : \mathsf{A}(A,B)} \text{ rec}$$

$$\frac{\begin{array}{c} \forall X \\ \left\{ \begin{array}{c} [\Gamma \mid E \xrightarrow{g} X] \\ \vdots \\ \hline \Gamma \mid a_i : A_i, y : E \mapsto t_i : G_i(X,B) \end{array} \right\}_{i=1,\ldots,k} \\ \hline [\Gamma \mid E \xrightarrow{g} \mathsf{A}(A,B)] \\ \vdots \\ \Gamma \mid \Delta \vdash s : Y \end{array}}{\Gamma \mid \Delta \vdash \mathsf{unfold}\ g(y)\ \mathsf{as}\ (\mathsf{D}_i : a_i.t_i)\ \mathsf{in}\ s : Y} \text{ unfold}$$

Figure 6: Coinductive typing rules

$$\begin{array}{lll} \mathsf{allnats} & = & \mathsf{unfold}\ g(x)\ \mathsf{as} \\ & & \quad \mathsf{Head} : x; \\ & & \quad \mathsf{Tail} : g(\mathsf{Succ}(x)) \\ & & \mathsf{in}\ g(\mathsf{Zero}) \end{array} \qquad \begin{array}{lll} \mathsf{map}(f|l) & = & \mathsf{unfold}\ g(l)\ \mathsf{as} \\ & & \quad \mathsf{Head} : \mathsf{Eval}(f, \mathsf{Head}(l)); \\ & & \quad \mathsf{Tail} : g(\mathsf{Tail}(l)) \\ & & \mathsf{in}\ g(l) \end{array}$$

Figure 7: Pola coinductive examples

$$\frac{g = \lambda y.(\cdots), \Gamma \vdash s \Rightarrow e}{\Gamma \vdash \mathsf{unfold}\ g(y)\ \mathsf{as}\ (\cdots)\ \mathsf{in}\ s \Rightarrow e}$$

$$\frac{}{\Gamma \vdash (\mathsf{D}_i : a_i.t_i) \Rightarrow (\Gamma \mid \mathsf{D}_i : a_i.t_i)}$$

$$\frac{\Gamma \vdash r \Rightarrow (\Gamma' \mid \mathsf{D}_i : a_i.t_i) \quad \Gamma \vdash s \Rightarrow e' \quad a_j = e', \Gamma' \vdash t_j \Rightarrow e}{\Gamma \vdash \mathsf{D}_j(s, r) \Rightarrow e}$$

Figure 8: Operational semantics: coinductive

Corecursive programs are defined using the unfold syntax, whose typing rule (the unfold box) is given in Figure 6. Some simple examples of programs which use the unfold syntax are given in Figure 7. The first creates an infinite list of natural numbers and the second applies a player function to each of the elements of an infinite list to create a new infinite list. The evaluation procedure for coinductive terms is lazy and is discussed in the next section.

4.6 PTIME soundness with coinductive types

Provided coinductive terms are evaluated lazily the system remains polynomial sound, as demonstrated by the proofs of the following two theorems. Before we state and prove the theorems it is necessary to extend our notion of normal form element to records with context $(\Gamma \mid \mathsf{D}_i : a_i.t_i)$. As not all records – e.g. infinite lists – can be interpreted as an $\mathbb{N}$-sized set, we shall simply decree that all records have size zero. Clearly this is an oversimplification as products and even higher-order functions can be assigned a meaningful size (recall $\mathbb{N}$-sized sets is affine closed with products). Nevertheless, to simplify the proofs, we shall only derive bounds for programs with inductive inputs.

In principle there does not seem to be any difficulty in extending the result for coinductive inputs; however, more bookkeeping is required as the bounds will now depend on the bounding functions involved with unfolding the data type. By assuming we have inductive inputs these are explicitly present and so no extra bookkeeping is necessary.

Before starting we need to extend our operational semantics to cover the coinductive terms. Here, terms are evaluated to "weak" normal form – weak in the sense that no reduction occurs within the body of a record. The rules are given in Figure 8.

Theorem 4.3 (size; inductive + coinductive) *Suppose* $x_1 : X_1, \ldots, x_n : X_n \mid y_1 : Y_1, \ldots, y_m :$

$Y_m \vdash u : Y$ is provable in the type system, where all input types are inductive. Then there exists a polynomial $\alpha[u](x_1, \ldots, x_n)$ such that for all normal form inputs $e_1 : X_1, \ldots, e_n : X_n$ and $a_1 : Y_1, \ldots, a_m : Y_m$, whenever $x_1 = e_1, \ldots, x_n = e_n, y_1 = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ in the operational semantics, $|e| \leq p(|e_1|, \ldots, |e_n|) + \sum_{i=1}^{m} |a_i|$.

Notational conventions: when the normal form terms are clear from the context we shall write $s \Rightarrow e$ instead of $\Gamma \vdash s \Rightarrow e$ and $|e| \leq p(\Gamma) + \Sigma$ instead of $|e| \leq p(|e_1|, \ldots, |e_n|) + \sum_{i=1}^{m} |a_i|$.

PROOF: We simply check the cases not covered in the previous theorem.

case: $u \equiv (\mathsf{D}_i : a_i.t_i)$ and $\Gamma \mid \Delta \vdash u : \mathsf{A}(A, B)$. As the size of a record is taken to be 0, the bound holds trivially in this case.

case: $u \equiv \mathsf{unfold}\ g(y)\ \mathsf{as}\ (\mathsf{D}_i : a_i.t_i)\ \mathsf{in}\ s$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : Y$. By the induction hypothesis, $s \Rightarrow e$ with $|e| \leq \alpha[s](\Gamma) + \Delta$, and so $u \Rightarrow e$ with the same bound.

case: $u \equiv \mathsf{D}_j(s, r)$ and $\Gamma \mid \Delta \vdash u : G_j(\mathsf{A}(A, B), B)$. Then $\Gamma \mid \Delta_1 \vdash s : A_j$ and $\Gamma \mid \Delta_2 \vdash r : \mathsf{A}(A, B)$, where $\Delta = \Delta_1, \Delta_2$. Then for any normal form inputs to Γ, Δ we have by the induction hypothesis, $s \Rightarrow e'$ with $|e'| \leq \alpha[s](\Gamma) + \Delta_1$, and $r \Rightarrow (\Gamma, \Delta_2 \mid \mathsf{D}_i : a_i.t_i)$. By the induction hypothesis we have that $t_j \Rightarrow e$ with $|e| \leq \alpha[t_j](\Gamma) + |e'| + \Delta_2$. Therefore:

$$\begin{aligned} |e| &\leq \alpha[t_j](\Gamma) + |e'| + \Delta_2 \\ &\leq \alpha[t_j](\Gamma) + \alpha[s](\Gamma) + \Delta_1 + \Delta_2 \end{aligned}$$

This exhausts the cases and finishes the proof. □

The time calculation with coinductive terms is a little more subtle because it is no longer the case that the computation inside of a box is constant time (in a fixed opponent context). So the time bound for the evaluation of a recursive function f must be revisited.

Theorem 4.4 (time; inductive + coinductive) *Let $\Gamma \mid \Sigma \vdash u : Y$, where $\Gamma \equiv x_1 : X_1, \ldots, x_n : X_n$ and $\Sigma \equiv y_1 : Y_1, \ldots, y_m : Y_m$, and all input types are inductive. Then there is a polynomial $\gamma[u](x_1, \ldots, x_n)$ with the property that for any normal form inputs, $e_1 : X_1, \ldots, e_n : X_n$ and $a_1 : Y_1, \ldots, y_m : Y_m$, the size of the evaluation tree of $x_i = e_i, y_i = a_i \vdash u \Rightarrow e$ is bounded by $\gamma[u](|e_1|, \ldots, |e_n|)$.*

Notational convention: when the normal form elements are clear from the context, we write $\|u\|_T$ to denote the size of the evaluation tree of $x_1 = e_1, \ldots, x_n = e_n, y_i = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ in the operational semantics.

PROOF: We argue the cases not covered in the inductive proof and revisit the the recursive function f case.

case: $u \equiv (\mathsf{D}_i : a_i.t_i)$ and $\Gamma \mid \Delta \vdash u : \mathsf{A}(A, B)$. In this case the term reduces to a record. As the body of the record is not evaluated until destruction, this is constant time.

case: $u \equiv \mathsf{unfold}\ g(y)\ \mathsf{as}\ (\mathsf{D}_i : a_i.t_i)\ \mathsf{in}\ g(s)$ and $\Gamma \mid \Delta \vdash u : \mathsf{A}(A, B)$. Then $\Gamma \mid \Delta \vdash s : E$ and for each i, $\Gamma \mid a_i : A_i, y : E \vdash t_i : G_i(\mathsf{A}(A, B), B)$. In this case:

$$\begin{array}{rcl} \|u\|_T & \leq & \|s\|_T + 1 \\ & \leq & \gamma[s](\Gamma) + \Delta + 1 \end{array}$$

case: $u \equiv \mathsf{D}_j(s, r)$ and $\Gamma \mid \Delta \vdash u : G_j(\mathsf{A}(A, B), B)$. Then $\Gamma \mid \Delta_1 \vdash s : A_j$ and $\Gamma \mid \Delta_2 \vdash r : \mathsf{A}(A, B)$, where $\Delta = \Delta_1, \Delta_2$. In this case $r \Rightarrow (\Gamma, \Delta_2 \mid \mathsf{D_i} : a_i.t_i)$ in constant time. By the induction hypothesis, $t_j \Rightarrow e$ in time bounded by $\gamma[t_j](\Gamma)$. Therefore:

$$\begin{array}{rcl} \|u\|_T & \leq & \|s\|_T + \|t_j\|_T + \|r\|_T + 1 \\ & \leq & \gamma[s](\Gamma) + \gamma[t_j](\Gamma) + k \end{array}$$

where k is a constant.

case: $u \equiv f(s, t)$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid \quad \vdash s : \mathsf{A}(B)$, $\Gamma \mid \Delta \vdash t : E$ and $\Gamma, x'_i : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X, B), y : E \vdash t_i : Y$. By the induction hypothesis, we have time bounds $\gamma[t_i](\Gamma, x'_i) + d_i$ where d_i is a polynomial which represents the *potential* time bound associated with the destruction of the coinductive terms. The point is that d_i is not constant! Initially, d_i is bounded by the potential time associated with t, but this potential can be increased by a polynomial $\gamma[t_i]^P(\Gamma, x'_i)$ with each evaluation of the box. If $s \Rightarrow e'$, then there can be at most $|e'|$ evaluations of the box, hence d_i is bounded by:

$$r_i := |e'| \cdot \max_i(\gamma[t_i]^P(\Gamma, |e'|)) + \gamma[t]^P(\Gamma)$$

So the time cost for one evaluation of the box is bounded by:

$$\max_i(\gamma[t_i](\Gamma, |e'|) + r_i)$$

and the time bound for the entire evaluation of f is given by:

$$\begin{array}{rcl} \|u\|_T & \leq & |e'| \cdot \max_i(\gamma[t_i](\Gamma, |e'|) + r_i) + \|s\|_T + \|t\|_T + 1 \\ & \leq & \alpha[s](\Gamma) \cdot \max_i(\gamma[t_i](\Gamma, \alpha[s](\Gamma)) + r_i) + \gamma[s](\Gamma) + \gamma[t](\Gamma) + 1 \end{array}$$

This exhausts the cases and finishes the proof. □

5 Bunched Pola

Pola's simple type system is reasonably expressive; however, there are nevertheless certain natural polynomial time programs, for example insertion sort, which can only be expressed in a machine level style of programming. To remedy this situation, Pola uses a construct called "peek": this is a case construct with a relaxed typing:

$$\frac{\Gamma \mid \Delta \vdash s : \mathsf{Bool} \quad \Gamma \mid \Delta \vdash t_1 : Y \quad \Gamma \mid \Delta \vdash t_2 : Y}{\Gamma \mid \Delta \vdash \mathsf{peek}\ s\ \mathsf{of}\ \{\mathsf{True}.t_1; \mathsf{False}.t_2\} : Y}$$

Peeking introduces non-affineness into the player world, as the context Δ appears both in the condition of the peek and in the bodies. The peek and the pcase have the same operational behaviour and the polynomial *size* bound remains intact for this new construct. Indeed, if $t_1 \Rightarrow e_1$ with $|e_1| \leq \alpha[t_1](\Gamma) + \Delta$ and $t_2 \Rightarrow e_2$ with $|e_2| \leq \alpha[t_2](\Gamma) + \Delta$, and $s \Rightarrow b$, then $\mathsf{peek}\ s\ \mathsf{of}\ \{\mathsf{True}.t_1; \mathsf{False}.t_2\} \Rightarrow e$ where $e \equiv e_1$ if $b \equiv \mathsf{True}$ and $e \equiv e_2$ if $b \equiv \mathsf{False}$. Thus:

$$|e| \leq \max(\alpha[t_1](\Gamma), \alpha[t_2](\Gamma)) + \Delta$$

However, the time bound is no longer polynomial as it is not the case that the number of evaluations of the fold box is bounded by the number of constructors in the subject of the recursion. Indeed, there is enough power to encode PSPACE complete problems as we now demonstrate.

5.1 PSPACE completeness and QSAT

The peek construct provides enough computational power to program PSPACE-complete problem: this is demonstrated by explicitly giving a Pola program for evaluating closed quantified boolean expressions, QSAT, a well-know PSPACE complete problem.

We begin by defining the data type of quantified boolean formulas:

$$\mathsf{data\ QBF} \to C = \begin{cases} \mathsf{T} : \mathbf{1} \to C \\ \mathsf{F} : \mathbf{1} \to C \\ \mathsf{And} : C, C \to C \\ \mathsf{Or} : C, C \to C \\ \mathsf{Not} : C \to C \\ \mathsf{All} : C \to C \\ \mathsf{Ex} : C \to C \\ \mathsf{Var} : \mathsf{Nat} \to C \end{cases}$$

The encoding $\langle _ \rangle$ of quantified boolean formulas is defined inductively by:

$$\begin{array}{rcl}
\langle \mathbf{t} \rangle & = & \mathsf{T} \\
\langle \mathbf{f} \rangle & = & \mathsf{F} \\
\langle x \rangle & = & \mathsf{Var}(\underline{n}) \\
\langle \phi \wedge \psi \rangle & = & \mathsf{And}(\langle \phi \rangle, \langle \psi \rangle) \\
\langle \phi \vee \psi \rangle & = & \mathsf{Or}(\langle \phi \rangle, \langle \psi \rangle) \\
\langle \neg \phi \rangle & = & \mathsf{Not}(\langle \phi \rangle) \\
\langle \exists x.\phi \rangle & = & \mathsf{Ex}(\langle \phi \rangle) \\
\langle \forall x.\phi \rangle & = & \mathsf{All}(\langle \phi \rangle)
\end{array}$$

where n is the number of quantifiers between the variable and the quantifier which binds it in the syntax tree. For example, the formula:

$$\phi = \exists x(\forall y(x \vee y) \wedge \forall z(z \wedge x))$$

is translated as:

$$\mathsf{Ex}(\mathsf{And}(\mathsf{All}(\mathsf{Or}(\mathsf{Var}(\underline{1}), \mathsf{Var}(\underline{0}))), \mathsf{All}(\mathsf{And}(\mathsf{Var}(\underline{0}), \mathsf{Var}(\underline{1})))))$$

We shall also require a boolean type with a fail state, $\mathsf{Bool}_{\text{fail}}$, and the following auxiliary program:

$$\begin{array}{rcl}
\mathsf{lookup}(x|y) & = & \mathsf{fold}\ f(\#, y)\ \mathsf{as} \\
& & \quad \mathsf{Zero.pcase}\ y\ \mathsf{of} \\
& & \quad\quad \mathsf{Nil.Fail} \\
& & \quad\quad \mathsf{Cons}(h,t).h \\
& & \quad \mathsf{Succ}(_|n).\mathsf{pcase}\ y\ \mathsf{of} \\
& & \quad\quad \mathsf{Nil.Fail} \\
& & \quad\quad \mathsf{Cons}(h,t).f(n,t) \\
& & \mathsf{in}\ f(x,y)
\end{array}$$

The program $\mathsf{qsat} : \mathsf{QBF}\ | \ \rightarrow \mathsf{Bool}_{\text{fail}}$ for evaluating a totally quantified boolean formula is defined as follows:

$$\begin{array}{rcl}
\mathsf{qsat}(x|\ ) & = & \mathsf{fold}\ f(\#, y)\ \mathsf{as} \\
& & \quad \mathsf{T.True} \\
& & \quad \mathsf{F.False} \\
& & \quad \mathsf{Var}(n|_).\mathsf{lookup}(n|y) \\
& & \quad \mathsf{And}(_|a_1, _|a_2).\mathsf{peek}\ f(a_1, y)\ \mathsf{of}\ \{\mathsf{True}.f(a_2, y); \mathsf{False.False}\} \\
& & \quad \mathsf{Or}(_|a_1, _|a_2).\mathsf{peek}\ f(a_1, y)\ \mathsf{of}\ \{\mathsf{True.True}; \mathsf{False}.f(a_2, y)\} \\
& & \quad \mathsf{Not}(_|a).\mathsf{pcase}\ f(a, y)\ \mathsf{of}\ \{\mathsf{True.False}; \mathsf{False.True}\} \\
& & \quad \mathsf{Ex}(_|a).\mathsf{peek}\ f(a, \mathsf{Cons}(\mathsf{True}, y))\ \mathsf{of}\ \{\mathsf{True.True}; \mathsf{False}.f(a, \mathsf{Cons}(\mathsf{False}, y))\} \\
& & \quad \mathsf{All}(_|a).\mathsf{peek}\ f(a, \mathsf{Cons}(\mathsf{True}, y))\ \mathsf{of}\ \{\mathsf{True}.f(a, \mathsf{Cons}(\mathsf{False}, y)); \mathsf{False.False}\} \\
& & \mathsf{in}\ f(x, \mathsf{Nil})
\end{array}$$

Notice that the peek construct is used repeatedly in this encoding.

5.2 Distributivity

In our affine setting, the peek rule, as stated above, is interpreted as a kind of distribution law, namely, the distribution of *products* over finite types $F_k \equiv \mathbf{1} + \cdots + \mathbf{1}$ (k times). It seems more natural to assume that in the player category products always distribute over coproducts, not just over the finite types. In this case, we demand that there is a purely player map:

$$\delta : X \times (A + B) \to (X \times A) + (X \times B)$$

inverse to the canonical one in the reserve direction. In intuitionistic logic with "coproducts-in-context" this rule is easy to prove:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{X \vdash X}{X, A \vdash X} \quad \dfrac{A \vdash A}{X, A \vdash A}}{X, A \vdash X \times A}}{X, A \vdash (X \times A) + (X \times B)} \quad \dfrac{\dfrac{\dfrac{X \vdash X}{X, B \vdash X} \quad \dfrac{B \vdash B}{X, B \vdash B}}{X, B \vdash X \times B}}{X, B \vdash (X \times A) + (X \times B)}}{X, A + B \vdash (X \times A) + (X \times B)}}{X \times (A + B) \vdash (X \times A) + (X \times B)}$$

In the affine setting a similar proof works by replacing $\times$ by $\otimes$ and it expresses the fact that tensor product distributes over the coproduct. Products, of course, can be added to the affine setting with coinductive data but this will not force a distributive law for products over coproducts

In order to express this distributive law type theoretically it is convenient to move to a bunched type system [22]. In the bunched setting, the player context is a bunched context defined as follows:

$$\Sigma ::= \emptyset \mid x : A \mid \Sigma, \Sigma \mid \Sigma; \Sigma$$

where ',' is interpreted as a tensor product and ';' is interpreted as a cartesian product (both are assumed to be associative and symmetric). The bunched type system also facilitates pattern-matching in Pola and allows Leivant-style data types to be defined, adding to the expressiveness of the language. Furthermore, as we shall show next, the bunched system is PSPACE sound.

5.3 The bunched type system

Pola's bunched type system extends the simple type system by incorporating the full distribution law for products over coproducts. This greatly increases the expressive power of the language and also facilitates pattern-matching. We prove that the system is PSPACE sound; in later sections we shall show that disallowing certain types, e.g. higher-order, from distributing over the coproduct cuts the system back down to PTIME.

In the bunched type system sequents have the form:

$$\Gamma \mid \Sigma \vdash t : A$$

where Σ is bunched and Γ is unchanged from the simple type system. The notation $\Sigma(\Delta)$ stands for a bunched context Σ containing the subtree Δ. The fold and function call rules as well as the typing rules for coinductive terms are the same as before (interpreting player contexts as bunched contexts) and so are not reproduced here. The bunched typing rules for the basic system are summarized in Figure 9.

$$\frac{x : A \in \Gamma \text{ or } \Sigma}{\Gamma \mid \Sigma \vdash x : A}\ \text{id} \qquad \frac{\Gamma \mid \Sigma(x : A) \vdash t : C}{\Gamma, x : A \mid \Sigma() \vdash t : C}\ \text{lift}$$

$$\frac{\Gamma \mid\ \vdash s : A \quad \Gamma, x : A \mid \Sigma \vdash t : C}{\Gamma \mid \Sigma \vdash t \ \mathsf{where}\ \{x = s\} : C}\ \text{cut}_o \qquad \frac{\Gamma \mid \Delta \vdash s : A \quad \Gamma \mid \Sigma(x : A; \Delta) \vdash t : C}{\Gamma \mid \Sigma(\Delta) \vdash t \ \mathsf{where}\ \{x := s\} : C}\ \text{cut}_p$$

$$\frac{\Gamma \mid \Sigma_1 \vdash t_1 : A \quad \Gamma \mid \Sigma_2 \vdash t_1 : B}{\Gamma \mid \Sigma_1, \Sigma_2 \vdash (t_1, t_2) : A \otimes B} \qquad \frac{\Gamma \mid \Delta \vdash s : A \otimes B \quad \Gamma \mid \Sigma(x : A, y : B; \Delta) \vdash t : C}{\Gamma \mid \Sigma(\Delta) \vdash \mathsf{peek}\ s\ \mathsf{of}\ (x, y).t : C}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : A \quad \Gamma \mid \Sigma \vdash t_2 : B}{\Gamma \mid \Sigma \vdash (t_1; t_2) : A \times B} \qquad \frac{\Gamma \mid \Delta \vdash s : A \times B \quad \Gamma \mid \Sigma(x : A; y : B; \Delta) \vdash t : C}{\Gamma \mid \Sigma(\Delta) \vdash \mathsf{peek}\ s\ \mathsf{of}\ (x; y).t : C}$$

$$\frac{\Gamma \mid \Delta \vdash t : F_i(B, \mathsf{A}(B))}{\Gamma \mid \Delta \vdash \mathsf{C}_i(t) : \mathsf{A}(B)} \qquad \frac{\Gamma \mid\ \vdash s : \mathsf{A}(B) \quad \{\Gamma, x_i : F_i(B, \mathsf{A}(B)) \mid \Sigma \vdash t_i : C\}_{i=1,\ldots,k}}{\Gamma \mid \Sigma \vdash \mathsf{case}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\} : C}$$

$$\frac{\Gamma \mid \Delta \vdash s : \mathsf{A}(B) \quad \{\Gamma \mid \Sigma(x_i : F_i(B, \mathsf{A}(B)); \Delta) \vdash t_i : C\}_{i=1,\ldots,k}}{\Gamma \mid \Sigma(\Delta) \vdash \mathsf{peek}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\} : C}$$

Figure 9: Bunched typing

Notice that product ";" is now a primitive of the language; however, products are still evaluated lazily. The typing of the peek rule needs some explanation. The context Δ appears both in the typing of the subject of the peek and in the typing of the branches of the peek. This means that a branch of the peek can either use the variables corresponding to the deconstructed term in the subject of the peek or the variables in the context Δ, both not both. This same typing has been incorporated into all of the peek rules and into the player cut rule as well.

A special case of a bunched context is the flat context consisting entirely of ","'s. It is straightforward to prove the following result:

Proposition 5.1 *Suppose a term t is well-typed in the simple type system. Then the term t^*, where* pcase *has been replaced by* peek*, is well-typed and has the same typing in the bunched system.*

The peek rule embodies the distribution law of products over coproducts:

$$\begin{aligned} \mathsf{dist}(|x) \quad = \quad & \mathsf{peek}\ \mathsf{P1}(x)\ \mathsf{of} \\ & \quad \mathsf{In0}(z).(\mathsf{P0}(x); z) \\ & \quad \mathsf{In1}(z).(\mathsf{P0}(x); z) \end{aligned}$$

where $x : X \times (A + B)$. This generalizes the previous system by allowing peeking on arbitrary inductive types, not just on the finite types.

Every Hofmann-style data type has an analogous Leivant-style one, obtained by replacing the tensor "," by a product ";" in the definition. For example, a Leivant tree is defined by:

$$\mathsf{data}\ \mathsf{LT}(A) \to X = \begin{cases} \mathsf{LLeaf} : A \to X \\ \mathsf{LNode} : X; X \to X \end{cases}$$

$$
\begin{array}{lll}
\mathsf{LTree}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
 & & \quad \mathsf{Zero.LLeaf(Zero)} \\
 & & \quad \mathsf{Succ}(_|n).\mathsf{LNode}(f(n);f(n)) \\
 & & \mathsf{in}\ f(x)
\end{array}
\qquad
\begin{array}{lll}
\mathsf{and}(|x_1;x_2) & = & \mathsf{peek}\ x_1\ \mathsf{of} \\
 & & \quad \mathsf{True}.x_2 \\
 & & \quad \mathsf{False.False}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{andLT}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
 & & \quad \mathsf{LLeaf}(b).b \\
 & & \quad \mathsf{LNode}(_|b_1;_|b_2).\mathsf{and}(b1;b2) \\
 & & \mathsf{in}\ f(x)
\end{array}
\qquad
\begin{array}{lll}
\mathsf{id}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
 & & \quad \mathsf{Zero.Zero} \\
 & & \quad \mathsf{Succ}(_|n).\mathsf{peek}\ f(n)\ \mathsf{of} \\
 & & \quad\quad \mathsf{Zero.Succ}(f(n)) \\
 & & \quad\quad \mathsf{Succ}(_).\mathsf{Succ}(f(n)) \\
 & & \mathsf{in}\ f(x)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{store}(x|) & = & \mathsf{fold}\ f(\#,a)\ \mathsf{as} \\
 & & \quad \mathsf{Nil}.a \\
 & & \quad \mathsf{Cons}(x,xs).\mathsf{peek}\ \mathsf{Eval}(a,\mathsf{Nil})\ \mathsf{of} \\
 & & \quad\quad \mathsf{Nil}.f(xs,\lambda v.\mathsf{Cons}(x,\mathsf{Eval}(a,v))) \\
 & & \quad\quad \mathsf{Cons}(_,_).f(xs,\lambda v.\mathsf{Cons}(x,\mathsf{Eval}(a,v))) \\
 & & \mathsf{in}\ f(l,\lambda x.x)
\end{array}
$$

Figure 10: Pola examples with bunched typing

Some example programs which are well-typed in the bunched type system are given in Figure 10. In the next section, we shall show that the bunched type system is PSPACE sound, so the above programs are exponential time, but polynomial space bounded. We have already seen that PSPACE complete programs can be programmed with peeking (on booleans) so the bunched type system is also complete for PSPACE computations.

5.4 PSPACE soundness

In this section we prove that Pola's (pure) full type system is PSPACE sound. Although the typing discipline now has bunched contexts, terms are evaluated according to the operational semantics already given in Figures 4 and 8 (recall that peek is evaluated in the same way as pcase). We should note however that the term $(t_1;t_2)$ is interpreted as a record $(\mathsf{P0} : t_1, \mathsf{P1} : t_2)$ and peek s of $(x_1;x_2).t$ is interpreted as $t[x_1/\mathsf{P0}(z), x_2/\mathsf{P1}(z)]$ where $z = s$. Recall that in the simple type system records were taken to have size 0 and meaningful bounds were obtained for inductive inputs only. In the full type system products are inductive – though still evaluated lazily – and are assigned a meaningful size as follows: If $t_1 \Rightarrow e_1$ and $t_2 \Rightarrow e_2$, then we define $|(t_1;t_2)| = \max(|e_1|,|e_2|)$.

Notational convention: let Σ be a bunched context consisting of variables $y_1 : Y_1, \ldots, y_m : Y_m$. Given normal form inputs $a_1 : Y_1, \ldots, a_m : Y_m$, we define $|\Sigma|$ inductively as follows:

$$
\begin{array}{rcl}
|\emptyset| & = & 0 \\
|y_i : Y_i| & = & |a_i| \\
|\Sigma_1, \Sigma_2| & = & |\Sigma_1| + |\Sigma_2| \\
|\Sigma_1; \Sigma_2| & = & \max(|\Sigma_1|, |\Sigma_2|)
\end{array}
$$

In this section we handle both the inductive and coinductive cases at once. We first prove that well-typed terms evaluate to a weak normal form which is polynomially size bounded:

Theorem 5.2 (size; bunches) *Suppose $x_1 : X_1, \ldots, x_n : X_n \mid \Sigma \vdash u : Y$ is provable in the bunched type system, where $y_1 : Y_1, \ldots, y_m : Y_m$ are the variables in Σ, and all inputs are inductive. Then there exists a polynomial $\alpha[u](x_1, \ldots, x_n)$ such that for all normal form inputs $e_1 : X_1, \ldots, e_n : X_n$ and $a_1 : Y_1, \ldots, a_m : Y_m$, whenever $x_1 = e_1, \ldots, x_n = e_n, y_1 = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ in the operational semantics, $|e| \leq p(|e_1|, \ldots, |e_n|) + |\Sigma|$.*

Proof: We argue by structural induction on u. The recursion and coinductive cases are exactly as before, so we only consider the cases that require a new argument. A note about the recursion case: the previous (size) arguments relied on the fact that the number of recursive calls in the evaluation of a recursive function was bounded by the size of the data type being recursed on. In the bunched setting, this is no longer the case as peeking allows both components of a product to be evaluated (and both components can have a recursive call). However, only one component of a product can contribute to the size change in a box so the previous bound remains valid.

case: $u \equiv t \text{ where } \{x := s\}$ and $\Gamma \mid \Sigma(\Delta) \vdash u : C$. Then $\Gamma \mid \Delta \vdash s : A$ and $\Gamma \mid \Sigma(x : A; \Delta) \vdash t : C$. By the induction hypothesis we have $s \Rightarrow e'$ with $|e'| \leq \alpha[s](\Gamma) + \Delta$ and $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma) + \Sigma(|e'|; \Delta)$. Then:

$$\begin{aligned} |e| &\leq \alpha[t](\Gamma) + \Sigma(|e'|; \Delta) \\ &\leq \alpha[t](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta) \\ &\leq \alpha[t](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta) \end{aligned}$$

case: $u \equiv \text{peek } s \text{ of } (x_1, x_2).t$ and $\Gamma \mid \Sigma(\Delta) \vdash \text{peek } s \text{ of } (x_1, x_2).t : Y$. Then $\Gamma \mid \Delta \vdash s : X_1 \otimes X_2$ and $\Gamma \mid \Sigma(x_1 : X_1, x_2 : X_2; \Delta) \vdash t : Y$. By the induction hypothesis we have $s \Rightarrow (e_1, e_2)$ with $|(e_1, e_2)| = |e_1| + |e_2| \leq \alpha[s] + \Delta$ and $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma) + \Sigma(|e_1| + |e_2|; \Delta)$. Then:

$$\begin{aligned} |e| &\leq \alpha[t](\Gamma) + \Sigma(|e_1| + |e_2|; \Delta) \\ &\leq \alpha[t](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta) \\ &\leq \alpha[t](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta) \end{aligned}$$

case: $u \equiv (t_1; t_2)$ and $\Gamma \mid \Sigma \vdash (t_1; t_2) : X_1 \times X_2$. Then $\Gamma \mid \Sigma \vdash t_1 : X_1$ and $\Gamma \mid \Sigma \vdash t_2 : X_2$. By the induction hypothesis, $t_1 \Rightarrow e_1$ with $|e_1| \leq \alpha[t_1](\Gamma) + \Sigma$, and $t_2 \Rightarrow e_2$ with $|e_2| \leq \alpha[t_2](\Gamma) + \Sigma$. Recall that $(t_1; t_2)$ is a record so it is already in (weak) normal form and:

$$\begin{aligned} |r| &\leq \max(|e_1|, |e_2|) \\ &\leq \max(\alpha[t_1](\Gamma), \alpha[t_2](\Gamma)) + \Sigma \end{aligned}$$

case: $u \equiv \text{peek } s \text{ of } (x_1; x_2).t$ and $\Gamma \mid \Sigma(\Delta) \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : X_1 \times X_2$ and $\Gamma \mid \Sigma(x_1 : X_1; x_2 : X_2; \Delta) \vdash t : Y$. By the induction hypothesis, $s \Rightarrow r$ with $|r| \leq \alpha[s](\Gamma) + \Delta$ and $t \Rightarrow e$ with $|e| \leq \alpha[t](\Gamma) + \Sigma(|r|; \Delta)$. Then:

$$\begin{aligned} |e| &\leq \alpha[t](\Gamma) + \Sigma(|r|; \Delta) \\ &\leq \alpha[t](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta) \end{aligned}$$

case: $u \equiv \mathsf{peek}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma(\Delta) \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : \mathsf{A}(B)$ and $\Gamma \mid \Sigma(x_i : F_i(\mathsf{A}(B), B); \Delta) \vdash t_i : Y$. By the induction hypothesis, $s \Rightarrow \mathsf{C}_j(e')$ with $|\mathsf{C}_j(e')| \leq \alpha[s](\Gamma)+\Delta$, and $t_j \Rightarrow e$ with $|e| \leq \alpha[t_j](\Gamma) + \Sigma(|e'|; \Delta)$. Then:

$$\begin{aligned} |e| &\leq \alpha[t_j](\Gamma) + \Sigma(|e'|; \Delta) \\ &\leq \alpha[t_j](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta) \\ &\leq \alpha[t_j](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta) \\ &\leq \max_i(\alpha[t_i](\Gamma)) + \alpha[s](\Gamma) + \Sigma(\Delta) \end{aligned}$$

This exhausts the cases and finishes the proof. □

Next we show that every well-typed Pola program is polynomial space bounded.

Theorem 5.3 (space; bunches) *Let $x_1 : X_1, \ldots, x_n : X_n \mid \Sigma \vdash u : Y$, where the $y_1 : Y_1, \ldots, y_m : Y_m$ are the variables of Σ, and all inputs are inductive. Then there is a polynomial $\beta[u](x_1, \ldots, x_n)$ with the property that for any normal form inputs, $e_1 : X_1, \ldots, e_n : X_n$ and $a_1 : Y_1, \ldots, a_m : Y_m$, the space required in the evaluation of $x_i = e_i, y_i = a_i \vdash u \Rightarrow e$ is bounded by $\beta[u](|e_1|, \ldots, |e_n|) + |\Sigma|$.*

Notational convention: when the normal form elements are clear from the context, we write $\|u\|_S$ to denote the space required to evaluate $x_1 = e_1, \ldots, x_n = e_n, y_i = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ in the operational semantics.

Proof: We argue by structural induction on u.

case: $u \equiv x$ is a variable. If $\Gamma, x : X \mid \Sigma \vdash x : X$, then $\beta[u](\Gamma, x) = x$; if $\Gamma \mid \Sigma(x : X) \vdash x : X$, then $\beta[u](\Gamma) = 0$.

case: $u \equiv t\ \mathsf{where}\ \{x := s\}$ and $\Gamma \mid \Sigma(\Delta) \vdash u : C$. Then $\Gamma \mid \Delta \vdash s : A$ and $\Gamma \mid \Sigma(x : A; \Delta) \vdash t : C$. If $s \Rightarrow e'$ then:

$$\begin{aligned} \|u\|_S &\leq \max(\beta[s](\Gamma) + \Delta, \beta[t](\Gamma) + \Sigma(|e'|; \Delta)) \\ &\leq \max(\beta[s](\Gamma) + \Delta, \beta[t](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta)) \\ &\leq \max(\beta[s](\Gamma) + \Delta, \beta[t](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta)) \\ &\leq \max(\beta[s](\Gamma), \beta[t](\Gamma) + \alpha[s](\Gamma)) + \Sigma(\Delta) \end{aligned}$$

case: $u \equiv t\ \mathsf{where}\ \{x = s\}$ and $\Gamma \mid \Sigma \vdash u : C$. Then $\Gamma \mid\ \vdash s : A$ and $\Gamma \mid \Sigma \vdash t : C$. If $s \Rightarrow e'$ then:

$$\begin{aligned} \|u\|_S &\leq \max(\beta[s](\Gamma), \beta[t](\Gamma, |e'|) + \Sigma) \\ &\leq \max(\beta[s](\Gamma), \beta[t](\Gamma, \alpha[s](\Gamma)) + \Sigma) \\ &\leq \max(\beta[s](\Gamma), \beta[t](\Gamma, \alpha[s](\Gamma))) + \Sigma \end{aligned}$$

case: $u \equiv (t_1, t_2)$ and $\Gamma \mid \Sigma \vdash u : X_1 \otimes X_2$, then $\Gamma \mid \Sigma_1 \vdash t_1 : X_1$ and $\Gamma \mid \Sigma_2 \vdash t_2 : X_2$, where $\Sigma = \Sigma_1, \Sigma_2$. If $t_1 \Rightarrow e_1$ and $t_2 \Rightarrow e_2$ then:

$$\begin{aligned} \|u\|_S &= \max(\|t_1\|_S + |e_2|, |e_1| + \|t_2\|_S) \\ &\leq \max(\beta[t_1](\Gamma) + \Sigma_1 + \alpha[t_2](\Gamma) + \Sigma_2, \alpha[t_1](\Gamma) + \Sigma_1 + \beta[t_2](\Gamma) + \Sigma_2) \\ &\leq \max(\beta[t_1](\Gamma) + \alpha[t_2](\Gamma), \alpha[t_1](\Gamma) + \beta[t_2](\Gamma)) + \Sigma_1 + \Sigma_2 \end{aligned}$$

case: $u \equiv (t_1; t_2)$ and $\Gamma \mid \Sigma \vdash u : X_1 \times X_2$, then $\Gamma \mid \Sigma \vdash t_1 : X_1$ and $\Gamma \mid \Sigma \vdash t_2 : X_2$. Recall that this is a coinductive record and that no computation takes place until destruction. The space requirements for a record, then, are the space required to store the context and a constant factor to store the unevaluated code. This gives a bound of the correct form.

case: $u \equiv \mathsf{peek}\ s\ \mathsf{of}\ (x_1, x_2).t$ and $\Gamma \mid \Sigma(\Delta) \vdash \mathsf{peek}\ s\ \mathsf{of}\ (x_1, x_2).t : Y$. Then $\Gamma \mid \Delta \vdash s : X_1 \otimes X_2$ and $\Gamma \mid \Sigma(x_1 : X_1, x_2 : X_2; \Delta) \vdash t : Y$. If $s \Rightarrow e'$, then:

$$\begin{aligned}
\|u\|_S &\leq \max(\|s\|_S, \beta[t](\Gamma) + \Sigma(|e'|; \Delta)) \\
&\leq \max(\beta[s](\Gamma) + \Delta, \beta[t](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta)) \\
&\leq \max(\beta[s](\Gamma) + \Delta, \beta[t](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta)) \\
&\leq \max(\beta[s](\Gamma), \beta[t](\Gamma) + \alpha[s](\Gamma)) + \Sigma(\Delta)
\end{aligned}$$

case: $u \equiv \mathsf{peek}\ s\ \mathsf{of}\ (x_1; x_2).t$ and $\Gamma \mid \Sigma(\Delta) \vdash \mathsf{peek}\ s\ \mathsf{of}\ (x_1; x_2).t : Y$. Then $\Gamma \mid \Delta \vdash s : X_1 \times X_2$ and $\Gamma \mid \Sigma(x_1 : X_1; x_2 : X_2; \Delta) \vdash t : Y$. Recall that u is short for the term $t[x_1/\mathsf{P0}(z), x_2/\mathsf{P1}(z)]$ where $\{z = s\}$. Supposing $\mathsf{P0}(s) \Rightarrow e_1$ and $\mathsf{P1}(s) \Rightarrow e_2$ then:

$$\begin{aligned}
\|u\|_S &\leq \beta[t](\Gamma) + \Sigma(|z|; \Delta) \\
&\leq \beta[t](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta) \\
&\leq \beta[t](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta)
\end{aligned}$$

case: $u \equiv \mathsf{C}_i(t)$ and $\Gamma \mid \Sigma \vdash u : \mathsf{A}(B)$. Then $\Gamma \mid \Sigma \vdash t : F_i(\mathsf{A}(B), B)$. Then:

$$\begin{aligned}
\|u\|_S &= \|t\|_S + 1 \\
&\leq \beta[t](\Gamma) + 1 + \Sigma
\end{aligned}$$

case: $u \equiv \mathsf{case}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma \vdash u : Y$. Then $\Gamma \mid\ \vdash s : \mathsf{A}(B)$ and $\Gamma, x_i : F_i(\mathsf{A}(B), B) \mid \Sigma \vdash t_i : Y$. Suppose $s \Rightarrow \mathsf{C}_j(e')$, then:

$$\begin{aligned}
\|u\|_S &\leq \max(\|s\|_S, \beta[t_j](\Gamma, |e'|) + \Sigma) \\
&\leq \max(\beta[s](\Gamma), \beta[t_j](\Gamma, \alpha[s](\Gamma))) + \Sigma \\
&\leq \max_i(\beta[s](\Gamma), \beta[t_i](\Gamma, \alpha[s](\Gamma))) + \Sigma
\end{aligned}$$

case: $u \equiv \mathsf{peek}\ s\ \mathsf{of}\ \{\mathsf{C}_i(x_i).t_i\}$ and $\Gamma \mid \Sigma(\Delta) \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : \mathsf{A}(B)$ and $\Gamma \mid \Sigma(x_i : F_i(\mathsf{A}(B), B); \Delta) \vdash t_i : Y$. If $s \Rightarrow \mathsf{C}_j(e')$, then:

$$\begin{aligned}
\|u\|_S &\leq \max(\|s\|_S, \beta[t_j](\Gamma) + \Sigma(|e'|; \Delta)) \\
&\leq \max(\beta[s](\Gamma) + \Delta, \beta[t_j](\Gamma) + \Sigma(\alpha[s](\Gamma) + \Delta; \Delta)) \\
&\leq \max(\beta[s](\Gamma) + \Delta, \beta[t_j](\Gamma) + \alpha[s](\Gamma) + \Sigma(\Delta)) \\
&\leq \max_i(\beta[s](\Gamma), \beta[t_i](\Gamma) + \alpha[s](\Gamma)) + \Sigma(\Delta)
\end{aligned}$$

case: $u \equiv \mathsf{fold}\ f(\#, y)\ \mathsf{as}\ \{\mathsf{C}_i(x_i'|x_i).t_i\}\ \mathsf{in}\ s$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid \Delta \vdash s : Y$ and:

$$\begin{aligned}
\|u\|_T &\leq \|s\|_T + 1 \\
&\leq \beta[s](\Gamma) + \Delta + 1
\end{aligned}$$

case: $u \equiv \mathsf{unfold}\ g(y)\ \mathsf{as}\ (\mathsf{D}_i : a_i.t_i)\ \mathsf{in}\ s$ and $\Gamma \mid \Delta \vdash u : \mathsf{A}(A, B)$. Then $\Gamma \mid \Delta \vdash s : \mathsf{A}(A, B)$ and:

$$\begin{aligned} \|u\|_T &\leq \|s\|_T + 1 \\ &\leq \beta[s](\Gamma) + \Delta + 1 \end{aligned}$$

case: $u \equiv f(s, t)$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid\ \vdash s : \mathsf{A}(B)$, $\Gamma \mid \Delta \vdash t : E$ and $\Gamma, x'_i : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X, B), y : E \vdash t_i : Y$. Suppose $s \Rightarrow \mathsf{C}_j(e')$ and $u \Rightarrow e$, then:

$$\begin{aligned} \|u\|_S &\leq |e| \cdot \max_i(\beta[t_i](\Gamma, |e'|)) + \|s\|_S + \|t\|_S \\ &\leq \alpha[s](\Gamma) \cdot \max_i(\beta[t_i](\Gamma, \alpha[s](\Gamma))) + \beta[s](\Gamma) + \beta[t](\Gamma) + \Delta \end{aligned}$$

case: $u \equiv (\mathsf{D}_i : a_i.t_i)$ and $\Gamma \mid \Delta \vdash u : \mathsf{A}(A, B)$. Then for each i, $\Gamma \mid a_i : A_i, \Delta \vdash t_i : G_i(\mathsf{A}(A, B), B)$. Again, this is a lazy coninductive record, and the argument is the same as the product case above.

case: $u \equiv \mathsf{D}_i(s, t)$ and $\Gamma \mid \Delta \vdash u : G_i(\mathsf{A}(A, B), B)$. Then $\Gamma \mid \Delta_1 \vdash s : A_i$ and $\Gamma \mid \Delta_2 \vdash t : \mathsf{A}(A, B)$, where $\Delta = \Delta_1, \Delta_2$. If $s \Rightarrow e'$, then:

$$\begin{aligned} \|u\|_S &\leq \max(\|s\|_S, \beta[t_i](\Gamma) + |e'| + \Delta_2) \\ &\leq \max(\beta[s](\Gamma) + \Delta_1, \beta[t_i](\Gamma) + \alpha[s](\Gamma) + \Delta_1 + \Delta_2) \\ &\leq \max(\beta[s](\Gamma), \beta[t_i](\Gamma) + \alpha[s](\Gamma)) + \Delta_1 + \Delta_2 \end{aligned}$$

This exhausts the cases and finishes the proof. □

As any PSPACE algorithm can be simulated in exponential time, we expect an exponential time upper bound for the full type system:

Theorem 5.4 (time; bunches) *Let $x : X_1, \ldots, x_n : X_n \mid \Sigma \vdash u : Y$, where $y_1 : Y_1, \ldots, y_m : Y_m$ are the variables in Σ, and all inputs are inductive. Then there is a polynomial $\gamma[u](x_1, \ldots, x_n)$ with the property that for any normal form inputs, $e_1 : X_1, \ldots, e_n : X_n$ and $a_1 : Y_1, \ldots, a_m : Y_m$, the size of the evaluation tree of $x_i = e_i, y_i = a_i \vdash u \Rightarrow e$ is bounded by $2^{\gamma[u](|e_1|, \ldots, |e_n|)}$.*

Notational convention: when the normal form elements are clear from the context, we write $\|u\|_T$ to denote the size of the evaluation tree of $x_1 = e_1, \ldots, x_n = e_n, y_i = a_1, \ldots, y_m = a_m \vdash u \Rightarrow e$ in the operational semantics.

Proof: The proof is by structural induction on the term u. This time we only give the interesting cases:

case: $u \equiv t\ \mathsf{where}\ \{x = s\}$ and $\Gamma \mid \Sigma \vdash u : C$. Then $\Gamma \mid\ \vdash s : A$ and $\Gamma, x : A \mid \Sigma \vdash t : C$. If $s \Rightarrow e'$ then:

$$\begin{aligned} \|u\|_T &\leq 2^{\gamma[s](\Gamma)} + 2^{\gamma[t](\Gamma, |e'|)} + 1 \\ &\leq 2^{\gamma[s](\Gamma) + \gamma[t](\Gamma, \alpha[s](\Gamma)) + 1} \end{aligned}$$

Notice the use of the (polynomial) size bound α here.

case: $u \equiv f(s,t)$ and $\Gamma \mid \Delta \vdash u : Y$. Then $\Gamma \mid \ \vdash s : \mathsf{A}(B)$, $\Gamma \mid \Delta \vdash t : E$ and $\Gamma, x'_i : F_i(\mathsf{A}(B), B) \mid x_i : F_i(X,B), y : E \vdash t_i : Y$. By the induction hypothesis, there are polynomials $\gamma[t_i](\Gamma, x'_i)$ such that $2^{\max_i(\gamma[t_i])}$ bounds the time to evaluate a box. Suppose $s \Rightarrow e'$. As both projections of a product may be evaluated, the number of recursive calls is bounded by $2^{|e'|}$, not by $|e'|$ as before. Therefore the number of evaluations of the box is bounded by:

$$\begin{aligned} \|u\|_T &\leq 2^{|e'|} \cdot 2^{\max_i(\gamma[t_i])} \\ &\leq 2^{\alpha[s](\Gamma)+\max_i(\gamma[t_i](\Gamma,\alpha[s](\Gamma)))} \end{aligned}$$

The other cases are treated similarly. □

5.5 ELEMENTARY soundness and completeness

If duplication is allowed in the player world then Hofmann-style data types and Leivant-style data types are isomorphic, and the exponential function 2^x can be defined by building an exponential size tree and counting its leaves. More explicitly, the program for building an exponential size tree is given by:

$$\begin{aligned} \mathsf{HTree}(x|) \ = \ & \mathsf{fold}\ f(\#)\ \mathsf{as} \\ & \quad \mathsf{Zero}.\mathsf{Leaf}(\mathsf{Zero}) \\ & \quad \mathsf{Succ}(_|n).\mathsf{Node}(f(n), f(n)) \\ & \mathsf{in}\ f(x) \end{aligned}$$

and the program for calculating the size of of a Hofmann tree is given by:

$$\begin{aligned} \mathsf{size}(x|) \ = \ & \mathsf{fold}\ f(\#, y)\ \mathsf{as} \\ & \quad \mathsf{Leaf}(n).\mathsf{Succ}(y) \\ & \quad \mathsf{Node}(_|l, _|r).f(l, f(r,y)) \\ & \mathsf{in}\ f(x, \mathsf{Zero}) \end{aligned}$$

Composing these programs gives a program for the function 2^x. So it is possible to program the functions in the set $\{n+m, n^2, rem(n,m), 2^n\}$. As the closure of this set under composition characterizes the class of Elementary time functions, we know that the resulting type system is complete for that complexity class.

On the other hand, we claim that we get no more than the Elementary time computable functions. As with all of the bounding arguments in this report, we first establish a bound on the size of the evaluated terms with respect to the sizes of the input arguments. A now familiar structural induction on terms shows that if $\Gamma \mid \Delta \vdash t : Y$ and $t \Rightarrow e$, then there exists a fixed height n such that:

$$|e| \leq \exp_2^n(|\Gamma|) \cdot |\Delta|$$

The important case in the argument is the bound on a recursive function $f(s,t)$. In this case, if $s \Rightarrow e'$, $t \Rightarrow e''$ and $f(s,t) \Rightarrow e$, then:

$$|e| \leq \exp_2^n(|\Gamma|)^{|e'|} \cdot |e''|$$

where $\exp_2^n(|\Gamma|)$ is the multiplicative factor bounding the size change in the evaluation of the box. This is a bound of the correct form. Another structural induction on terms shows that if $\Gamma \mid \Delta \vdash u : Y$, then there exists a fixed height n such that:

$$\|u\|_T \leq \exp_2^n(|\Gamma|)$$

Thus we have established:

Theorem 5.5 *Pola's full type system with the assumption that the affine tensor and the product in the p-world are identified is sound and complete for the ELEMENTARY complexity class.*

Obviously, if the tensor and product are isomorphic, then the player category is cartesian closed and hence distributive.

6 PTIME within PSPACE

The model of computation that we have developed here strictly contains the traditional model of computation in that there are many new types, given by data declaration (both inductive and coinductive), which are not traditionally considered. These new types allow a number of distinctions which are not available in the traditional setting. In particular, the relationship between PTIME and PSPACE is much sharper in this setting due to the presence of lazily evaluated coinductive data. In fact, as we shall discuss there is an immediate separation between PTIME and nondeterministic polynomial time (NP). This section opens with a discussion of these matters and the manner in which "nondeterminism" is interpreted in the present setting.

The section ends on a more practical note: namely on the business of cutting down bunched Pola to PTIME. Clearly, it is not possible to provides a system which detects *exactly* when a PSPACE program actually runs in PTIME, thus, any such system is going to involves some pragmatic choices. Here we suggest a simple system, which obtains a significant increase in expressive power over the simply typed Pola. It works by simply disallowing contraction of coinductive and universal types. This immediately means that the system strictly includes the simply typed system (which as it was affine never allowed contraction) – so it is PTIME complete. Inevitably, it also disallows some programs which obviously have a PTIME behavior.

6.1 Nondeterminism

Recall that the size of a Leivant tree is its height (the maximum size of its branches). This is because, in the simply typed system of Pola, one is prevented from exploring both branches of such a tree as one can only affinely perform one projection. This intuitively means, that, in the simple type system of Pola, a Leivant tree can only be used by generating a series of choices of projections in order to reach a leaf. The series of choices that this entails determines a path in the tree and this we shall as a "certificate" to surmount the nondeterminism which is represented by the tree.

In the (pure) full type system of Pola, by way of contrast, one can explore both branches of a Leivant tree as demonstrated by the program for evaluating the disjunction of a Leivant tree of booleans:

$$
\begin{array}{rcl}
\mathsf{OrLT}(x|) & = & \mathsf{fold}\ f(\#)\ \mathsf{as} \\
 & & \quad \mathsf{LLeaf}(b).b \\
 & & \quad \mathsf{LNode}(z).\mathsf{peek}\ f(\mathsf{P0}(z))\ \mathsf{of} \\
 & & \quad\quad \mathsf{True}.\mathsf{True} \\
 & & \quad\quad \mathsf{False}.f(\mathsf{P1}(z)) \\
 & & \mathsf{in}\ f(x)
\end{array}
$$

Note that to achieve this the universal variable z has been distributed by the peek: it is this which produces the exponential number of recursive calls and whence an exponential time (albeit PSPACE) program.

It is significant that any program for calculating the disjunction of a Leivant tree of booleans must have an exponential time behavior as, in the worse case, the program must inspect *every* leaf of the tree. The point is that a program written for an arbitrary Leivant tree cannot know what value a given leaf will take unless it forces the evaluation of that leaf to discover the value. On the other hand any program which *never* evaluates a particular leaf cannot possibly compute the disjunction correctly as it will not be able to distinguish between the tree which is false everywhere

and the tree which is true only at that leaf (both these Leivant trees can be easily constructed in the simple type system for Pola).

To make things more precise we also have to consider algorithms which, depending on values of certain leaves, will inspects certain other leaves; such an algorithm may on any particular value never inspects all the leaves, yet there may be no leaf which the algorithm *never* evaluates. However, by considering the tree which is everywhere false any such algorithm cannot fail to evaluate a leaf of this tree as if it did we would then be free to flip the value of that leaf – to make it true – without affecting the output (which would mean the algorithm did not implement OrLT). Therefore on this tree the program is forced to spend exponential time. Indeed, every algorithm for OrLT must visit all the leaves in a particular order: however, clearly the minute it finds a True leaf it cannot do better than to declare immediately that the answer is True. This means that even the average running time of the algorithm must be exponential.

Thus we may conclude:

Proposition 6.1 *There is no PTIME program in the pure full type system of Pola which implements* OrLT*.*

We should pause here to say what we actually mean by a PTIME program in bunch typed Pola. We mean a program which, when supplied with arguments, has its running time (in the sense used here) bounded by a polynomial in the sizes of its arguments. We are guaranteed that this includes all programs in simply typed Pola, however, by serendipity or for other reasons (e.g. as described shortly) there could be many others. Thus, PTIME is being used here in an essentially standard way.

We now wish to argue that OrLT is a nondeterministic polynomial time (NP) predicate: to do this we shall follow the spirit of the presentation in [3] in which the idea of "leaf complexity" was introduced. Given a tree t which evaluates to True under OrLT one can easily produce a "certificate" by modifying the program:

```
CertOrLT  ::  LTree(Bool)|  ⟶ SF(BNat)
  (x| )    =  fold f(#) as
                 LLeaf(True).SS(End)
                 LLeaf(False).FF
                 LNode(z).peek f(P0(z)) of
                    SS(n).SS(B0(n))
                    FF.peek f(P1(z)) of
                       SS(n).SS(B1(n))
                       FF.FF
              in f(x)
```

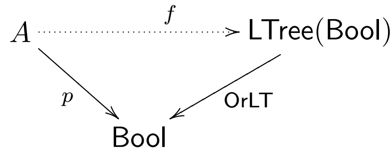
As the program which produces the certificate is in PSPACE the size of the certificate is automatically polynomially bounded. Notice that the program actually produces more than a "certificate": it either indicates that there is no path which ends at True by returning FF, or that there is such a path which it provides as a binary natural number n by returning SS(n). Given such a certificate it is then easy to produce a PTIME program – actually one in the simple Pola type system – which

verifies that the tree has True at that leaf:

$$
\begin{array}{rcl}
\mathsf{CheckOrLT} & :: & \mathsf{SF(BNat)|LTree(Bool)} \longrightarrow \mathsf{SF(Bool)} \\
(\mathsf{FF}|t) & = & \mathsf{SS(False)} \\
(\mathsf{SS}(n)|t) & = & \mathsf{fold}\ f(\#,t)\ \mathsf{as} \\
& & \quad \mathsf{B0}(n).\mathsf{pcase}\ t\ \mathsf{of} \\
& & \quad\quad \mathsf{LLeaf}(b).\mathsf{FF} \\
& & \quad\quad \mathsf{LNode}(t).f(n,\mathsf{P0}(t)) \\
& & \quad \mathsf{B1}(n).\mathsf{pcase}\ t\ \mathsf{of} \\
& & \quad\quad \mathsf{LLeaf}(b).\mathsf{FF} \\
& & \quad\quad \mathsf{LNode}(t).f(n,\mathsf{P1}(t)) \\
& & \quad \mathsf{End}.\mathsf{pcase}\ t\ \mathsf{of} \\
& & \quad\quad \mathsf{LLeaf(True).SS(True)} \\
& & \quad\quad \mathsf{LNode}(_).\mathsf{FF} \\
& & \mathsf{in}\ f(x|t)
\end{array}
$$

More precisely what the verifying program does is to report the predicate is successfully False, returning SS(False), when no path is provided by the certificate finder and otherwise it checks that the path does actually lead to a True leaf, returning SS(True). If the certificate makes no sense it returns FF.

Thus OrLT is, in this sense, an NP problem. In fact, we may define $p : A \longrightarrow \mathsf{Bool}$ in Pola to be an NP problem precisely when there is a PTIME factorization of p through OrLT:



Thus, OrLT is by defnition an NP-complete problem. To see that this is sensible observe that to specify a nondeterministic polynomial time Turing machine for a problem is (following the ideas in [3]) exactly to specify a PTIME map to LTree(Bool). The path to a leaf provides the string of bits for the PTIME Turing machine which constitutes the computation to deternine whether the leaf is true or false.

This discussion is to support the observation:

Corollary 6.2 *In bunched* Pola*,* PTIME $\neq$ NP*.*

While not wishing to minimize the importance of this observation, it is worth pointing out that this is *not* the classical P versus NP problem which is quite tightly specified on the binary natural numbers (or lists of booleans). Indeed, a little bit of thought will make one realize that SAT, in Pola, is no longer even an NP-complete problem! This is because, in particular, OrLT cannot be reduced to it. So, not all PTIME settings are equal! Here it is the presence of coinductive data which allows this easy distinction. Of course, arguably a reasonable polynomial time programming setting *should* support coinductive data!

6.2 The polynomial hierarchy in Pola

It seems reasonable to ask, given the above, whether the whole polynomial hierachy (provably) separates in Pola. We believe that this is the case and, here, discuss why this might be expected. In Pola the generic coNP-complete problem is given by

```
AndLT(x|)  =  fold f(#) as
                 LLeaf(b).b
                 LNode(z).peek f(P0(z)) of
                    False.False
                    True.f(P1(z))
              in f(x)
```

A certificate for this coNP problem is, of course, a path to a false leaf which demonstrates the predicate is false. There is no PTIME program in Pola which can turn a coNP-complete problem into a NP problem as, in particular, such would mean that there would be a PTIME map f such that

$$\begin{array}{ccc} \mathsf{LTree(Bool)} & \overset{f}{\dashrightarrow} & \mathsf{LTree(Bool)} \\ & \searrow^{\mathsf{AndLT}} \quad \swarrow_{\mathsf{OrLT}} & \\ & \mathsf{Bool} & \end{array}$$

However, if f sets even a single leaf of the codomain tree True it must inspect *all* the leaves of its domain as they must all be true. This immediately means that f in the worst case must be exponential and so there is no PTIME reduction.

Corollary 6.3 *In bunched* Pola, NP $\neq$ coNP*.*

We may now define inductively:

$$\begin{array}{ccc} \mathsf{LTree}(\mathsf{LTree}^n(\mathsf{Bool})) & \xrightarrow{\mathsf{LTree}(\mathsf{AndLT}^n)} & \mathsf{LTree(Bool)} \\ & \searrow^{\mathsf{OrLT}^{n+1}} & \downarrow \mathsf{OrLT} \\ & & \mathsf{Bool} \end{array}$$

$$\begin{array}{ccc} \mathsf{LTree}(\mathsf{LTree}^n(\mathsf{Bool})) & \xrightarrow{\mathsf{LTree}(\mathsf{OrLT}^n)} & \mathsf{LTree(Bool)} \\ & \searrow^{\mathsf{AndLT}^{n+1}} & \downarrow \mathsf{AndLT} \\ & & \mathsf{Bool} \end{array}$$

with $\mathsf{OrLT}^0 = \mathsf{AndLT}^0 = 1 : \mathsf{Bool} \longrightarrow \mathsf{Bool}$. We claim that OrLT^{n+1} is the complete proposition for Σ_n and AndLT^n is the complete proposition for Π_n in the polynomial hierarchy. Furthermore, that there is no PTIME reduction from one to the other for much the same reasons as above: to know that one should have one of the leaves of the bottom tree in the codomain evaluate to False involves knowing that *all* the arguments of the bottom tree in the domain evaluate to False, but this already involves inspecting exponentially many things. This we claim suffices to show that the hierarchy does not collapse.

Usually Δ_{n+1} is defined by adding an oracle from Π_n or Σ_n: however, in the context of the type systems we have been describing the notion of "adding" an oracle does not have an obvious interpretation. Indeed, if one simply adds OrLT as a p-map to the system one obtains a system in which the whole hierarchy is included!

6.3 Cutting down to PTIME

Returning to the problem of how to recognize functions such as OrLT as non-PTIME, notice that the universal type (of the fold) is used in both the condition of the peek and in the body of the False branch: this means it has been duplicated (a contraction over semicolon) and then used in both the condition and a branch of a peek (categorically a distribution of a product over a coproduct). This suggests that, in cutting down to PTIME, one should somehow disallow this combination of duplication and distribution for universal types. However, this is certainly not enough by itself to secure PTIME soundness. For example, consider the following program:

$$\begin{array}{lcl} \mathsf{hdup}(x|) & = & \mathsf{fold}\ f(\#, y)\ \mathsf{as} \\ & & \quad \mathsf{Zero.Eval}(y, \mathsf{True}) \\ & & \quad \mathsf{Succ}(_|n).f(n, \lambda v.\mathsf{and}(\mathsf{Eval}(y, \mathsf{True}); \mathsf{Eval}(y, \mathsf{False}))) \\ & & \mathsf{in}\ f(x, \lambda v.v) \end{array}$$

In this case, the universal variable n is not duplicated or distributed. However, the higher-order parameter y is duplicated and then used as arguments to the and function. The and function, of course, has hidden in it a distribution which forces the evaluation of both its arguments. This allows the fold to build up an exponential time computation which is released when the Zero branch of the program is reached. In this case this unwanted behaviour is caused by the duplication of the higher-order variable y and the distribution hidden in the and function.

In fact, these two cases essentially capture *all* the ways one can get outside polynomial time: one must, inside a fold, duplicate a term which has a potentially non-constant time cost. The only types which can exhibit such behavior are universal and coinductive types. The difficulty is to arrange that no such combination of duplication and distribution happens. It may seem that the best approach to this is to control distributions – as this is where the computational damage actually happens – however, it is easier to control the duplication end of the problem.

Below we sketch a strategy for cutting down the bunched Pola system to PTIME.

Let us call a type which does not contain any coinductive or universal types **distributing**. Note, for example, if X is a universal type, $A \otimes X$ is not distributing even if A is distributing. Then one way to control this undesirable behavior is by insisting that non-distributing types cannot be duplicated. Thus what we want to do ultimately is to mark types which are duplicated as they can be filled only with distributing types. To do this we must be able to detect duplications. A technique for doing this is as follows, annotate the proofs of our system from the leaves downward. There are two sorts of annotation: we shall use a **hat**, $\widehat{A}$, to mean the type has never been accessed (has been weakened) and a **bar**, $\overline{A}$, to mean the type has been duplicated (and so must be distributing). Note that annotated type is not a new type. Thus, the annotation always applies to the whole type.

The first stage is to annotate the system to indicate which parts of the context are actually used. This process starts at the leaves of the proof of the term where variables are accessed or constants are created. In the former case the id rule is annotated with all the types hatted *except*

for the type from which the variable is extracted. In the latter case all the types in the context are hatted. As contexts are combined the hat annotation is only propagated if all the (premise) contexts have a hat on that type.

At rules which contain a contraction (i.e. all the peek rules, and the p-cut) we annotate the fact that a type has been duplicated by a bar: duplication only happens when the contraction is between two non-hatted types. For example for the peek rule:

$$\frac{\Gamma \mid \ \vdash s : A_1 + A_2 \quad \Gamma \mid \Sigma(x_1 : A_1; (x : A, y : \widehat{B})) \vdash t_1 : C \quad \Gamma \mid \Sigma(x_2 : A_2; (x : A, y : B))) \vdash t_2 : C}{\Gamma \mid \Sigma(x : \overline{A}, y : B) \vdash \mathsf{peek}\ s\ \mathsf{of}\ \{\sigma_1(x_1).t_1; \sigma_2(x_2).t_2\} : C}$$

One must then process the remaining rules in the same manner. Peeking on a product always causes duplication of the subject variable (besides the expected duplication due to the contraction of the context) unless all but one of the components are not used. Note that the p-cut rule also contains some contraction and so can result in some barring of types. The fact that the subject of the cut is barred has no effect: beyond the requirement that the barred type is distributing.

We remark that there is apparently no point in barring types as we can replace barring by simply inspecting them to see that they are distributing. However, in the presence of a parametric polymorphic typed system and function definitions one can use the barred types to ensure type substitutions do not break the barring constrains.

A term whose proof can be successfully annotated in this manner is a cut-down Pola program. We make two simple observations:

Lemma 6.4 *Every simply typed* Pola *program is a cut-down* Pola *program.*

Proof: A simply typed proof always gives a bunched Pola proof. However, such a proof never results in any barring as the system is affine. □

Theorem 6.5 (time; cut-down bunches) *Let* $\Gamma \mid \Sigma \vdash u : Y$*, be a well-typed cut-down* Pola *program with all inputs are inductive. Then there is a polynomial* $\gamma[u](x_1, \ldots, x_n)$ *with the property that for any normal form inputs,* $e_1 : X_1, \ldots, e_n : X_n$ *and* $a_1 : Y_1, \ldots, a_m : Y_m$*, the size of the evaluation tree of* $x_i = e_i, y_i = a_i \vdash u \Rightarrow e$ *is bounded by* $\gamma[u](|e_1|, \ldots, |e_n|)$*.*

Proof: (Sketch) We first note that the α bound remains correct.

The proof proceeds as before by a structural induction on terms.

The only nonstandard case is that of a recursive function $f(s,t)$ defined by a fold. In that case, the argument rests on preventing a non-distributing type within the context of a fold from being duplicated and thus allowing the possibility that one cannot bound the computation with a polynomial. With this restriction in place, however, the proof proceeds exactly as before. □

7 Conclusion

This document describes the categorical semantics and type systems underlying lower complexity systems. In particular it shows how various complexity classes (PTIME, PSPACE, and ELEMENTARY) can be characterized implicitly in this family of type systems.

These type systems underly the development of a PTIME programming language called Pola, a polymorphic functional style language. Notably this document does not attempt to explain how type inference is achieved in this language. It is notable that the programming language which emerges from is surprisingly expressive. However, this document gives little indication of how this expressiveness plays out in practice nor does are the limits of the expressiveness of these various systems explored in any depth.

An interesting aspect of these settings is that they do provide a native notion of equality of maps because the (inductive and coinductive) data types come packaged with the universal properties. Thus, this means the initial settings come with a term logic with a native notion of equality and it would be interesting to know how this relates to the various logics [14] which have been considered already for expressing PTIME and PSPACE. Certainly this expands the scope of the study of these lower complexity systems to include program equivalence which, a priori, cannot be assumed to be the same as in more powerful systems.

Also of interest is the behavior of the complexity classes within PSPACE: we have argued that in this semantics the polynomial hierarchy separates. While we have not been able to see any direct implication for the classical P versus NP problem, it is natural to wonder whether there may be. Certainly, this semantics shows that separation can be achieved not only with oracles but also by moving to richer type systems. Certainly, a striking aspect of this system is that it is a quite natural interpretation of what PTIME programming should encompass.

References

[1] J-M. Andreoli. Focussing and proof construction. *Ann. Pure and Applied Logic*, 107(1) (2001) 131–163.

[2] S. Bellantoni & S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[3] D. Bovet & P. Crescenzi & R. Silvestri. A uniform approach to define complexity classes. *Theoretical Computer Science*, 104 (1992) 263–283.

[4] M.J. Burrell. Pola project page. `http://projects.wizardlike.ca/projects/pola`.

[5] R. Cockett & B. Redmond. A categorical setting for lower complexity. To appear in *Electronic Notes in Theoretical Computer Science.*

[6] R. Cockett & R. Seely. Polarized category theory, modules and game semantics. *Theory and Application of Categories*, 18 (2007) 4–101.

[7] R. Cockett & D. Spencer. Strong Categorical Datatypes I. *International Meeting on Category Theory 1991*, AMS, Canadian Mathematical Society Proceedings, 1992.

[8] J-Y Girard. A new constructive logic: classical logic. *Mathematical Structures in Computer Science*, 1(3) (1991) 255–296.

[9] M. Hamano & P. Scott. A categorical semantics for Polarized MALL. *Annals of Pure and Applied Logic*, 145 (2007) 276-313.

[10] C. Hermida & B. Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145(2) (1998) 107–152.

[11] M. Hofmann & U. Dal Lago. Quantitative Models and Implicit Complexity. FSTTCS 2005: Proceedings of Foundations of Software Technology and Theoretical Computer Science, Hyderabad, India. *Springer Lecture Notes in Computer Science*, 3821 (2005) 189–200.

[12] M. Hofmann. Type systems for polynomial-time computation. Habilitation thesis, University of Darmstadt, 1999.

[13] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1) (2003) 57-85.

[14] Neil Immerman, Guest Column: *Progress in Descriptive Complexity*, SIGACT NEWS 36(4) (2005) 24-35.

[15] A. Kock. Strong Functors and Monoidal Monads. *Archiv der Math.* 23: 113120, 1972.

[16] Olivier Laurent. Étude de la polarisation en logique, Université Aix-Marseille II, Thèse de Doctorat (2002).

[17] Olivier Laurent. Polarized Games. *Annals of Pure and Applied Logic* no 1–3, Vol. 130 (2004) 79–123.

[18] D. Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, (1993) 325-333.

[19] D. Leivant & J.-Y. Marion. Ramified Recurrence and Computational Complexity II: Substitution and Poly-space. In *Proc. CSL'94, Springer LNCS*, 933 (1994) 486-500.

[20] J. Otto. *Complexity Doctrines*. Ph.D. thesis, McGill University, 1995.

[21] R. Paré & L. Román. Monoidal categories with natural numbers object. *Studia Logica*, 48(3) (1989) 361 – 376.

[22] P. W. O'Hearn & D.J. Pym. The logic of bunched implications. *Bull. Symbolic Logic*, Volume 5, Number 2 (1999), 215 – 244.

[23] L. Santocanale. A calculus of circular proofs and its categorical semantics. *Proc. FOSSACS 2002, Springer LNCS*, No. 2303, 2002.

[24] V. Vene. *Categorical programming with inductive and coinductive types.* PhD thesis, University of Tartu, 2000.

[25] Richard J. Wood, *Indicial methods for relative categories.* PhD Thesis, Dalhousie University, 1976.