

Pola: a language for PTIME programming

Michael J. Burrell¹ and Robin Cockett² and Brian F. Redmond²

¹ Computer Science Department, University of Western Ontario,
London, Ontario, Canada, N6A 5B7

² Department of Computer Science, University of Calgary,
Calgary, Alberta, Canada, T2N 1N4

Abstract. This paper describes the type system of a functional style programming language called Pola which is complete with respect to polynomial time programming. This means, both that every well-typed Pola program is guaranteed to halt in time polynomial with respect to the size of its input, and that all such polynomial time functions can be written in Pola.

The type system is polarized and supports two sorts of data: player (or safe) data and opponent (or normal) data. Opponent programs have polynomial complexity, while player programs are restricted to be constant time with respect to the opponent context. The system supports a wide range of inductive and (higher-order) coinductive data types. A form of safe recursion is used to limit the power of programming with the inductive types which, it is argued, results in a surprisingly natural programming style. Pola’s coinductive types are forced by the type system to have constant time destruction: an aspect which may, quite independently, be of interest for real-time programming.

The focus of the paper is Pola’s type system and the proof that it is complete with respect to polynomial time programming. Examples are given to illustrate both the programming style of Pola and how Pola’s type system maintains polynomial complexity.

1 Introduction

Pola is a functional style programming language whose type system guarantees that all its programs run in polynomial time. Indeed, as every polynomial time algorithm can be rendered in Pola, the language is complete for polynomial time programming. Pola’s type system, which is the main focus of this paper, enforces a style of programming which, to someone familiar with functional programming, is—relative to what is being achieved—quite natural. Below we shall illustrate, with some Pola programs some aspects of the language and its type system.

Pola’s type system was inspired by the realization that Bellantoni and Cook’s system of safe recursion [1] can be viewed as the proof theory of a polarized logic [4]. Pola’s type system is influenced by this connection and as, in turn, polarized logics were developed to model games, the description of Pola’s type system inherits some game theoretic terminology. In particular, rather than speaking of “safe” and “normal” types we shall speak of “player” and “opponent” types or

worlds. In the game theoretic view it is the opponent who drives the computation while the player responds in constant time (and space).

To place this work in context it is useful to make some historical remarks about the development of *implicit* systems for PTIME programs. Bellantoni and Cook’s system of safe recursion is a simplification of the slightly earlier system of Leivant [9]. That system used tiered recursion and, in particular, supported a general class of inductive data. Bellantoni and Cook, besides replacing the system of tiers with safe recursion, also abandoned general inductive data in favour of modelling binary numbers. This limits their system as a basis for a programming language.

Hofmann, in his habilitationsschrift [7], aware of all the developments above, developed a modal type system for polynomial time programs which he modelled in a presheaf topos. Hofmann introduced a number of innovations. In particular, he used constant time affine computations as the basis for stepping up to polynomial time. Here we take a similar approach: Pola’s player world is both affine and populated by constant time computations (in a fixed opponent context).

Hofmann also considered how to translate Bellantoni’s and Cook’s system into his system. However, there was a significant mismatch as safe computation freely allows duplication (which is forbidden in an affine type system). To circumvent this problem Hofmann, rather arbitrarily, noted that allowing duplication of binary numbers does not actually disturb his system.

However, this really was a rather delicate issue. For example, as Hofmann himself observed, duplicating trees certainly lets super-polynomial computations into his system. So why are binary numbers so special? Even more curiously, for trees (and other inductive data) Hofmann’s system appeared to directly conflict with the approach of Leivant and Marion (which allowed duplication). Below we shall indicate how, in our type system, the difference between these views of inductive data are resolved.

Another curiosity of Hofmann’s system is his recursion schemes: while he provided specific examples he never presented a general form for recursion over inductive data. Here we do provide uniform recursion schemes both for inductive and coinductive data. Pola’s recursion scheme derives from the circular proof systems of Luigi Santocanale [12], which also appeared earlier (with other schemes) in Varmo Vene’s thesis [13]. The recursion scheme is interesting as it has some built-in higher-order content which allows one to avoid Colson’s objection [6, 11] to first-order recursion schemes.

Note there are features of Pola and the Pola implementation which we do not describe in this paper. In particular, there is a full type inference algorithm for Pola programs and an operational semantics which essentially is a by-value reduction of terms to head normal form (in which all coinductive data is left undestructed).

Finally we should remark on why we feel that developing a polynomial time language is of some interest (see also Caseiro’s discussion in [2]). The original motivation behind Pola was, in fact, driven by the investigation of implicit type systems for low complexity. The implementation of Pola has definitely enhanced

our understanding of these systems. However, we do not believe that this is purely of theoretical interest. Pola programs are both time and space bounded and it is possible to actually calculate these bounds. For applications in which these resources are an issue, programming in a system like Pola may have some real value. Furthermore, the “player” programs in Pola all require constant time and space. This makes them of particular interest in real-time systems and for processes in general. The coinductive types of Pola, which potentially provide the basis for process programming, live in this constant time world, so that their destruction always requires constant time (and space).

1.1 A brief introduction to syntax and semantics

Pola syntax is summarized in Appendix A. Pola is, in many respects, a typical functional language with call-by-value semantics. The most notable restriction is that general recursion is disallowed. Calls to functions previously defined are allowed, and described as “cuts” in the logic of safety, but generally recursion, either direct or indirect is disallowed.

“Safe recursion”, in the form of `fold` and `unfold` constructs will be dealt with in section 2.

Another major visible feature of the language is the splitting of variables into two worlds, the opponent world and the player world. To introduce the syntax, consider the bounded subtraction function:

$$\begin{aligned} \mathbf{monus} = n \mid m. \text{fold } f(x, y) \text{ as } \{ \\ \text{Zero. Zero;} \\ \text{Succ}(z). \text{case } y \text{ of } \{ \\ \text{Zero. } n; \\ \text{Succ}(r). \mathbf{Succ}(f(z, r)) \} \\ \} \text{ in } f(n, m) \end{aligned}$$

The type of the function would be $\mathbf{monus} : \mathbf{Nat} \mid \mathbf{Nat} \mapsto \mathbf{Nat}$. The vertical bar between the two parameters signifies that it takes one parameter in the opponent world (everything to the left of the vertical bar) and one parameter in the player world (everything to the right of the bar).

The player world typically disallows duplication, i.e., a player variable can only be referenced once. The opponent world allows duplication and, further, anything which is used to drive a recursion, such as the variable n above, is required to be in the opponent world.

Variables can be brought from the opponent world to the player world, but not the other way around. Variables introduced by `fold` and `case` constructs (e.g., the variables z and r in the example above) are necessarily in the player world. The precise mechanics of these constructs are dealt with in section 2.

In addition to allowing inductive data types, Pola allows coinductive data types, useful for describing infinite data structures, higher-order functions and products.

2 The proof theory of safe recursion

The sequents in the proof system we shall develop have the form:

$$\Gamma \mid \Delta \mapsto Y$$

They have a split antecedent which is a sequence of opponent types, Γ , followed by a sequence of player types, Δ ; the consequent is a player type Y . A sequent with an empty opponent context, Γ , is called a purely player/safe sequent, while a sequent with an empty player context, Δ , is called a purely opponent/normal sequent. Sequents with non-empty opponent and player contexts are called “cross sequents”. These sequents will be annotated to show how terms are formed. Each type in an antecedent will be annotated with a variable (for basic types) or a pattern for constructed types such as products. Variables cannot be repeated in an antecedent as they are intended to replace positional information: throughout we assume that antecedent sequences can be permuted freely (so the logic has exchange rules assumed). On the right hand side terms of the consequent type are formed.

2.1 The basic logic of safety

The judgments in table 1 describe the basic features of the logic of safety.

The opponent and player has identity sequents of each atomic type and the basic structural rule of weakening is present in either context as shown in table 1. Notice that we use the term variable x and the type variable X for the opponent and the term variable y and the type variable Y for the player.

The logic has both player and opponent cut rules: these are represented in the terms uniformly as (explicit) substitutions in table 1. The logic does not have explicit rules for contraction as in this formulation contraction is implemented by cutting with a weakened identity sequent, as shown below:

$$\frac{\frac{\Gamma, x : X \mid \mapsto x : X}{\Gamma, x : X \mid \Delta \mapsto t[x/x'] : Y} \text{id}_o \quad \Gamma, x : X, x' : X \mid \Delta \mapsto t : Y}{\Gamma, x : X \mid \Delta \mapsto t[x/x'] : Y} \text{cut}_o$$

Notice that contraction cannot be implemented on the player side as the player world is affine.

Another feature of the basic logic of safety is the ability to shift a type in the player context across to the opponent side. We do not have an explicit rule for this, however, such a “lifting” operation can be implemented with a player cut, as shown below:

$$\frac{\frac{\Gamma, u : Y \mid \mapsto u : Y}{\Gamma, u : Y \mid \Delta \mapsto t[u/y] : Y} \text{id}_o \quad \frac{\Gamma \mid \Delta, y : Y \mapsto t : Y'}{\Gamma, u : Y \mid \Delta, y : Y \mapsto t : Y'} \text{weak}_o}{\Gamma, u : Y \mid \Delta \mapsto t[u/y] : Y} \text{cut}_p$$

It is not possible in this logic to shift a type in the opponent context across to the player side.

$\frac{}{\Gamma, x : X \mid \Delta \mapsto x : X} \text{id}_o$	$\frac{}{\Gamma \mid \Delta, y : Y \mapsto y : Y} \text{id}_p$	
$\frac{\Gamma \mid \Delta \mapsto t : Y}{\Gamma, x : X \mid \Delta \mapsto t : Y} \text{weak}_o$	$\frac{\Gamma \mid \Delta \mapsto t : Y}{\Gamma \mid \Delta, y : Y' \mapsto t : Y} \text{weak}_p$	
$\frac{\Gamma \mid \mapsto t : X \quad \Gamma, x : X \mid \Delta \mapsto t' : Y}{\Gamma \mid \Delta \mapsto t'[t/x] : Y} \text{cut}_o$		
$\frac{\Gamma \mid \Delta_1 \mapsto t : Y \quad \Gamma \mid \Delta_2, y : Y \mapsto t' : Y'}{\Gamma \mid \Delta_1, \Delta_2 \mapsto t'[t/y] : Y'} \text{cut}_p$		
$\frac{\Gamma \mid \Delta \mapsto t : Y}{\Gamma, () : \mathbf{1} \mid \Delta \mapsto t : Y}$	$\frac{\Gamma \mid \Delta \mapsto t : Y}{\Gamma \mid \Delta, () : \mathbf{1} \mapsto t : Y}$	$\frac{}{\Gamma \mid \Delta \mapsto () : \mathbf{1}}$
$\frac{\Gamma, x_1 : X_1, x_2 : X_2 \mid \Delta \mapsto t : Y}{\Gamma, (x_1, x_2) : X_1 \otimes X_2 \mid \Delta \mapsto t : Y}$	$\frac{\Gamma \mid \Delta, y_1 : Y_1, y_2 : Y_2 \mapsto t : Y}{\Gamma \mid \Delta, (y_1, y_2) : Y_1 \otimes Y_2 \mapsto t : Y}$	
$\frac{\Gamma \mid \Delta_1 \mapsto t_1 : Y_1 \quad \Gamma \mid \Delta_2 \mapsto t_2 : Y_2}{\Gamma \mid \Delta_1, \Delta_2 \mapsto (t_1, t_2) : Y_1 \otimes Y_2}$		

Table 1. The basic logic of safety.

Next we have the rules for “products”, which for player types recall is an affine tensor. Product formation on the left produces a *pattern*, consisting of a pair of variables or an empty tuple. Product formations can occur in both opponent and player contexts, as shown in table 1. In addition there is a pairing rule which allow the formation of tuples of terms, as shown in table 1. Observe that the logic does not have an explicit term for projection, however, one can form a term for a projection:

$$\frac{\frac{\frac{}{| y_1 : Y_1 \mapsto y_1 : Y_1} \text{id}_p}{| y_1 : Y_1, y_2 : Y_2 \mapsto y_1 : Y_1} \text{weak}_p}{| (y_1, y_2) : Y_1 \otimes Y_2 \mapsto y_1 : Y_1}}$$

Substituting into the pattern (y_1, y_2) term produces the effect of projection.

It is possible to add coproducts explicitly to the system, but in order to keep the rules to a minimum, we shall instead treat them as a special case of inductive data (see Section 2.2).

The term logic allows a succinct way of expressing the cut elimination rewriting rules. These rules, in fact, express the substitution for terms of this logic, as shown in figure 1. Notice that the last rule indicates there is a confluence problem: which component of a pair is substituted first should not matter. Clearly, as these rules ultimately eliminate the explicit substitutions, this order does not matter and forces some program equivalences.

$$\begin{aligned} x[t/x'] &\Longrightarrow x & x \neq x' \\ x[t/x] &\Longrightarrow t \\ (t_1, t_2)[t/x] &\Longrightarrow (t_1[t/x], t_2[t/x]) \\ ()[t/x] &\Longrightarrow () \\ t[()] &\Longrightarrow t \\ t[(t_1, t_2)/(x_1, x_2)] &\Longrightarrow (t[t_1/x_1])[t_2/x_2] = (t[t_2/x_2])[t_1/x_1] \end{aligned}$$

Fig. 1. Substitution rules.

Although the term logic, at this stage, is not very expressive it does enjoy a simple cut elimination procedure which corresponds exactly to the natural rules for substitution, as shown in figure 1. The only slight subtlety is to realize that terms which can substitute patterns, due to the way the term are constructed, must themselves be patterned in the same manner.

Proposition 1. *The basic logic of safety enjoys cut-elimination.*

Proof. (Sketch.) We follow the techniques found in [5]. It is straightforward to show that the family of rewrites in figure 1 is confluent and always terminates.

2.2 Inductive data types

When inductive data is declared, which may be parameterized over a list of player types A , a new player type $D(A)$ is added, together with rules for its constructors C_i (see table 2). An inductive data type is introduced into the basic

$\frac{\Gamma \mid \Delta \mapsto t : F_i(D(A), A)}{\Gamma \mid \Delta \mapsto C_i(t) : D(A)} \text{ cons}_i$				
$\frac{\{ \Gamma \mid y_i : F_i(D(A), A), \Delta \mapsto t_i : Y \}_{i=1, \dots, k}}{\Gamma \mid y : D(A), \Delta \mapsto \text{case } y \text{ of } \left\{ \begin{array}{c} C_1(y_1).t_1 \\ \vdots \\ C_k(y_k).t_k \end{array} \right\} : Y} \text{ case}$				
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">$\Gamma, \forall C$</td> <td style="padding: 5px;">$f : \mid C, \tilde{E} \mapsto Y$</td> </tr> <tr> <td colspan="2" style="text-align: center; padding: 10px;"> $\left\{ \frac{\vdots}{y'_i : F_i(D(A), A) \mid y_i : F_i(C, A), \tilde{w} : \tilde{E} \mapsto t_i : Y} \right\}_{i=1, \dots, k}$ </td> </tr> </table>	$\Gamma, \forall C$	$f : \mid C, \tilde{E} \mapsto Y$	$\left\{ \frac{\vdots}{y'_i : F_i(D(A), A) \mid y_i : F_i(C, A), \tilde{w} : \tilde{E} \mapsto t_i : Y} \right\}_{i=1, \dots, k}$	
$\Gamma, \forall C$	$f : \mid C, \tilde{E} \mapsto Y$			
$\left\{ \frac{\vdots}{y'_i : F_i(D(A), A) \mid y_i : F_i(C, A), \tilde{w} : \tilde{E} \mapsto t_i : Y} \right\}_{i=1, \dots, k}$				
$\Gamma, z' : D(A) \mid \tilde{w}' : \tilde{E} \mapsto \text{fold } f(z, \tilde{w}) \text{ as } \left\{ \begin{array}{c} C_1(y'_1 @ y_1).t_1 \\ \vdots \\ C_k(y'_k @ y_k).t_k \end{array} \right\} \text{ in } f(z', \tilde{w}') : Y$				

Table 2. Judgements for fixed points and safe inductive folding.

logic of safety with a specification of the form:

$$\text{data } D(A) \rightarrow C = \{C_i : F_i(C, A) \mapsto C\}_{i=1, \dots, k}$$

where each F_i must be a *polarized functor*. The only polarized functors in this system are generated by constant functors K^A , \otimes , \times , and coproducts.

Example data types are the (unary) natural numbers, the binary numbers, lists, trees, booleans and coproducts. Their specification is illustrated in figure 2. In fact, any “polynomial” data type (constructed from sums and products) can be created. More examples will be provided after we have introduced coinductive data types (Section 2.5).

Each inductive data type comes equipped with a finite number of constructors which are used to generate instances of the data type. The rules for construction together with the case rule express the fixed point properties of inductive data types, as shown in table 2.

$$\begin{array}{lll}
\text{data Bool}() \rightarrow C & \text{data BNat}() \rightarrow C & \text{data T}(A) \rightarrow C \\
= \{ \text{True} : \mathbf{1} \mapsto C & = \{ \text{Empty} : \mathbf{1} \mapsto C & = \{ \text{Leaf} : A \mapsto C \\
\quad \text{False} : \mathbf{1} \mapsto C \} & \text{B0} : C \mapsto C & \quad \text{Node} : C \otimes C \mapsto C \} \\
& \text{B1} : C \mapsto C \} & \\
\\
\text{data L}(A) \rightarrow C & \text{data Nat}() \rightarrow C & \text{data } A + B \rightarrow C \\
= \{ \text{Nil} : \mathbf{1} \mapsto C & = \{ \text{Zero} : \mathbf{1} \mapsto C & = \{ \text{Inj0} : A \mapsto C \\
\quad \text{Cons} : C \otimes A \mapsto C \} & \text{Succ} : C \mapsto C \} & \quad \text{Inj1} : B \mapsto C \}
\end{array}$$

Fig. 2. Examples of inductive data type definitions.

2.3 Safe recursion

In addition to the fixed point rules we have an inductive rule which allows one to do a limited (or safe) recursion on an opponent data type. To express this we use a “recursion box” which binds universally the fixed point type (see table 2). The proof inside the box must, to maintain polynomial time, proceed entirely in the fixed opponent context Γ and so it is stored away at the top of the box. The type C *must* be universally quantified, i.e., it must not be able to match with any other type. Thus, it is universally quantified at the top of the box. The function f exists only within the fold and has its first parameter of the universally quantified type C .

Recursion over an inductive data type is driven by a opponent instance of the data type. However, the phrases of the fold, which correspond to the constructors, must be player programs. These player programs are allowed to use an opponent copy of the data over which it is currently recursing. The @ symbol binds this data to the variable which is driving the phrase of the fold. To allow use of the opponent context inside the box without modifying it for subsequent recursive calls, we introduce the following rule:

$$\frac{\Gamma', x : X \mid \Delta \mapsto t : Y}{\Gamma' \mid \Delta \mapsto t[x'/x] : Y} \quad (\text{provided } x' : X \in \Gamma)$$

Examples of safe recursions are given in figure 3. The definition of **add** is a simple illustration of the use of the fold construct. The fold introduces a function symbol, in this case f , which (safely) recurses over its first argument, x . It does this by pattern matching on x dealing with the case of x being **Zero** or **Succ** and in the latter case recursively calling f . Note also that the second argument of f has player type so cannot be passed as context. The multiplication, by way of contrast, has both arguments opponent types and the fold carries the second argument in as context.

If one were to attempt to program an exponential function in Pola, for example as:

exp = $n, m \mid \cdot \text{fold } f(x) \text{ as } \{ \mathbf{Zero.Succ}(\mathbf{Zero}); \mathbf{Succ}(z).\mathbf{mul}(m, f(z)) \}$ in $f(n)$

then there would be a type mismatch as **mul** expects both arguments to be of opponent type but the output of the recursively defined function, f , is always a

```

add = n | m.fold f(x, y) as
{ Zero.y;
  Succ(z).Succ(f(z, y)) }
in f(n, m)

sumList = l | .fold f(x) as
{ Nil.Zero;
  Cons(z'@z, zs).add(z', f(zs)) }
in f(l)

mul = n, m | .fold f(x) as
{ Zero.Zero;
  Succ(z).add(m, f(z)) }
in f(n)

sumTree = t | .fold f(x, y) as
{ Leaf(z'@z).add(z', y);
  Node(l, r). f(l, f(r, y)) }
in f(t, Zero)

oMap{f : A |  $\mapsto$  B} = x' | .fold h(x) as
{ Nil.Nil;
  Cons(a'@a, z).Cons(f(a'), h(z)) }
in h(x')

```

Fig. 3. Examples of Pola programs using safe recursion.

player type. Writing an exponential function on unary numbers is impossible in Pola.

The fold construct in the definition of **sumList** works similarly, pattern-matching and recursing, but over lists instead of natural numbers. A more complex example, **sumTree**, which illustrates how the fold construct has some built-in higher-order content, is given by summing the numbers at the leaves of a tree. The program traverses the tree accumulating the sum of the leaves. The simpler, though less efficient, program which passes up the sum of subtrees to be recursively added at the nodes does not type in Pola as the **add** function must have one argument of opponent type. The last example, **oMap**, shows that the List type constructor is functorial in the opponent world: notice how in this and the previous example we have to use the @ construct to obtain an opponent decomposition of the type.

2.4 Peeking

The intention behind the **peek** construct is that it should operate exactly like a case construct with the exception that we are not allowed to use the result of the peek beyond determining which constructor the object peeked on matches. For instance, we may peek to find out if a natural number is zero or non-zero, but are not allowed to find out its actual value. The trade-off is that the subject of the peek, what is being peeked on, is allowed to be referred to again.

As an example, imagine we wish to construct a function applies some other function, f , to its argument, x , only if $x > 0$. We can write such a function using a case:

```

iffNotZero = | x.case x of {
  Zero.Zero;
  Succ(y).f(Succ(y)) }

```

This works, but requiring the programmer to use `Succ(y)` instead of simply `x` speaks to an awkwardness and inefficiency. Peeks allow a more efficient implementation:

$$\mathbf{fIfNotZero} = | x.\mathbf{peek} \ x \ \mathbf{of} \ \{ \\ \mathbf{Zero}.\mathbf{Zero}; \\ \mathbf{Succ}(-).f(x) \}$$

This allows more freedom to the programmer in some cases while still ensuring polynomial time. Peeks do offer ability to do things which cases cannot. For instance, consider a function which adds its two parameters only if `x` is greater than `y`:

$$\mathbf{addIfBigger} = x | y.\mathbf{peek} \ gt(x, y) \ \mathbf{as} \ \{ \\ \mathbf{False}.\mathbf{Zero}; \\ \mathbf{True}.\mathbf{add}(x, y) \}$$

To do this using a case construct would require `y` to be an opponent variable instead of a player variable.

The peek rule, as shown in table 3, is the one exception to the affineness of

$$\boxed{\frac{\boxed{\Gamma | \Delta \mapsto t_0 : D(A)} \quad \{ \Gamma | \Delta \mapsto t_i : Y \}_{i=1, \dots, k}}{\Gamma | \Delta \mapsto \mathbf{peek} \ t_0 \ \mathbf{of} \ \left\{ \begin{array}{c} C_1(-).t_1 \\ \vdots \\ C_k(-).t_k \end{array} \right\} : Y}} \text{ peek}$$

Table 3. Judgement for peeking.

the player world as the player context can be reused. It is required to satisfy two crucial restrictions to avoid super-polynomial computations.

The first restriction ensures that a peek is a “pure” case: that is one can only use *anonymous variables* in the matched patterns. This means a peek, unlike a case, cannot pass deconstructed data into its body.

The second restriction is a more subtle scope restriction. One is not allowed to use any function symbol introduced by an enclosing (safe) recursion in the term which is the subject of a peek. To indicate this restriction we have placed a semi-impervious box around the peek hypothesis to block enclosing fold functions.

As the bounding argument will show, the second requirement is not necessary to maintain the polynomial space bound. Removing this scope restriction would make Pola complete for PSPACE computations as following [10] and [8] one can then program PSPACE complete problems (e.g. quantified Boolean formulas) in Pola. This was explored by Hofmann, in a different setting, in [8], where he introduced a slightly different notion called *restricted duplication*. The condition in Pola is not a type restriction, but rather a scope restriction, which leads to a slightly cleaner view of this phenomenon.

$$\boxed{
\begin{array}{c}
\frac{\Gamma \mid \Delta \mapsto t : \mathsf{D}(A/B) \otimes A_i}{\Gamma \mid \Delta \mapsto \mathsf{D}_i(t) : G_i(\mathsf{D}(A/B), B)} \text{ dest}_i \\
\\
\frac{\{ \Gamma \mid \Delta, a_i : A_i \mapsto t_i : G_i(\mathsf{D}(A/B), B) \}_{i=1\dots k}}{\Gamma \mid \Delta \mapsto \left(\begin{array}{c} \mathsf{D}_1 : a_1.t_1 \\ \vdots \\ \mathsf{D}_k : a_k.t_k \end{array} \right) : \mathsf{D}(A/B)} \text{ record} \\
\\
\frac{\Gamma, \forall C \quad g : \mid \tilde{E} \mapsto C}{\left\{ \frac{\vdots}{\mid a_i : A_i, \tilde{y} : \tilde{E} \mapsto t_i : G_i(C, B)} \right\}_{i=1,\dots,k}} \\
\\
\Gamma \mid \tilde{y}' : \tilde{E} \mapsto \text{unfold } g(\tilde{y}) \text{ as } \left(\begin{array}{c} \mathsf{D}_1 : a_1.t_1 \\ \vdots \\ \mathsf{D}_k : a_k.t_k \end{array} \right) \text{ in } g(\tilde{y}') : \mathsf{D}(A/B)
\end{array}
}$$

Table 4. Judgements for destruction, records and safe coinductive unfolding.

to a corresponding destructor, together with a value for the variable a_i , that computation proceeds by evaluating the appropriate term t_i .

Example Pola programs using coinductive types are given in figure 4. The type of the function **pMap** is $A \multimap B, \mathsf{L}(A) \mid \mapsto \mathsf{L}(B)$ and it simply applies the player function f to every element of the list. This shows that the list function is functorial over player functions, using the higher-order features of the language. However, recall that more generally we have already seen that $\mathsf{L}(A)$ is functorial over opponent functions – although this could not be programmed in a higher order function. The function **allNats** generates, via an **unfold**, the infinite list of all natural numbers.

3 Polynomial time completeness

The term logic admits an operational semantics in which programs are evaluated on inputs. The inputs are elements (maps from the final object) usually in the opponent category and they are evaluated against terms of the logic of safe recursion which represent programs. The observation we wish to make is that given a particular program there is a polynomial in the size of its inputs which bounds the running time. This is the sense in which the setting can represent at most polynomial time computations. For the reverse implication, namely that every polynomial time computation can be represented it is a simple matter to

code up a Turing machine and to run it for a polynomial length of time on its inputs. Thus, all polynomial time computations can be represented.

Only a rough sketch of the proof (for the inductive case) will be given here; the operational semantics together with a detailed proof of polynomial time soundness will be presented in the full version of the paper.

Polynomials can be assigned to terms inductively by the following rules. In fact, the derivation of the polynomial associated to a term mimics the derivation of the term by the proof rules.

$$\begin{aligned}
\llbracket x \rrbracket &= x && \text{(opponent variable)} \\
\llbracket y \rrbracket &= 0 && \text{(player variable)} \\
\llbracket (t_1, t_2) \rrbracket &= \llbracket t_1 \rrbracket + \llbracket t_2 \rrbracket \\
\llbracket t[t'/x] \rrbracket &= \llbracket t \rrbracket[\llbracket t' \rrbracket/x] && \text{(opponent cut)} \\
\llbracket t[t'/y] \rrbracket &= \llbracket t \rrbracket + \llbracket t' \rrbracket && \text{(player cut)} \\
\llbracket () \rrbracket &= 0 \\
\llbracket C(t) \rrbracket &= \llbracket t \rrbracket + 1 \\
\llbracket \text{case } y \text{ of } \{C_i(y'_i @ y_i).t_i\} \rrbracket &= \max_i \llbracket t_i[y/y'_i] \rrbracket \\
\llbracket \text{peek } y \text{ of } \{C_i(-).t_i\} \rrbracket &= \max_i \llbracket t_i \rrbracket \\
\llbracket \text{fold } f(z, \tilde{w}) \text{ as } \{\dots\} \text{ in } f(z', \tilde{w}') \rrbracket &= \llbracket f(z', \tilde{w}') \rrbracket \\
\llbracket f(z, \tilde{w}) \rrbracket &= z \cdot \llbracket \text{case } z \text{ of } \{C_i(y'_i @ y_i).t_i\} \rrbracket^* \\
\llbracket \text{unfold } g(\tilde{y}) \text{ as } (\dots) \text{ in } g(\tilde{y}') \rrbracket &= 0 \\
\llbracket (\dots) \rrbracket &= 0 && \text{(record)} \\
\llbracket D(t, t') \rrbracket &= \{t\} + \llbracket t \rrbracket + \llbracket t' \rrbracket && \text{(destruction)}
\end{aligned}$$

where $\{-\}$ is identical to $\llbracket - \rrbracket$ except that:

$$\begin{aligned}
\llbracket \text{unfold } g(\tilde{y}) \text{ as } (D_1 : a_1.t_1; \dots; D_k : a_k.t_k) \text{ in } g(\tilde{y}') \rrbracket &= \max_i \llbracket t_i \rrbracket^* \\
\llbracket (D_1 : a_1.t_1; \dots; D_k : a_k.t_k) \rrbracket &= \max_i \llbracket t_i \rrbracket \\
\llbracket D(t, t') \rrbracket &= \{t\}
\end{aligned}$$

where $\llbracket - \rrbracket^*$ has the same inductive definition as $\llbracket - \rrbracket$ except that $\llbracket f(z, \tilde{w}) \rrbracket^* = 0$ for each recursive call to f . One can show by induction on the proof rules that if $x_1 : X_1, \dots, x_n : X_n \mid y_1 : Y_1, \dots, y_m : Y_m \mapsto t : Y$ is derivable then $\llbracket t \rrbracket$ is a polynomial in x_1, \dots, x_n . One can then show that whenever t evaluates to e ,

$$|e| \leq \llbracket t \rrbracket + |y_1| + \dots + |y_m|$$

For example, addition on unary numbers is defined in Pola by $x' : \text{Nat} \mid y' : \text{Nat} \mapsto t : \text{Nat}$, where t is short for the term

$$\text{fold } f(x, y) \text{ as } \{\text{Zero}.y; \text{Succ}(z).\text{Succ}(f(z, y))\} \text{ in } f(x', y')$$

It is easy to check that $\llbracket t \rrbracket = x'$ and so the size of the output of the addition function is bounded by $|x'| + |y'|$, as expected. Once the size bound has been established it is relatively easy to show that all evaluation trees have polynomially bounded size on their inputs. This demonstrates rather conservatively that all evaluations can be bounded by a polynomial.

Intuitively, $\llbracket t \rrbracket$ can be thought of as the computational cost of t and $\{t\}$ can be thought of as the *potential* computational cost of t , only used for coinductive terms. As an example, consider $\mathbf{Head}(\mathbf{Tail}(\mathbf{Tail}(\mathbf{allNats})))$ where $\mathbf{allNats}$ is as defined in Figure 4.

$$\begin{aligned}
& \llbracket \mathbf{Head}(\mathbf{Tail}(\mathbf{Tail}(\mathbf{allNats}))) \rrbracket \\
&= \{ \mathbf{Tail}(\mathbf{Tail}(\mathbf{allNats})) \} + \llbracket \mathbf{Tail}(\mathbf{Tail}(\mathbf{allNats})) \rrbracket \\
&= \{ \mathbf{Tail}(\mathbf{allNats}) + (\{ \mathbf{Tail}(\mathbf{allNats}) \} + \llbracket \mathbf{Tail}(\mathbf{allNats}) \rrbracket) \} \\
&= \{ \mathbf{allNats} \} + (\{ \mathbf{allNats} \} + (\{ \mathbf{allNats} \} + \llbracket \mathbf{allNats} \rrbracket)) \\
&= 3\{ \mathbf{allNats} \} + \llbracket \mathbf{allNats} \rrbracket \\
&= 3\{\mathbf{unfold } g(x) \text{ as } (\mathbf{Head} : x; \mathbf{Tail} : g(\mathbf{Succ}(x))) \text{ in } g(\mathbf{Zero})\} + \llbracket \mathbf{unfold } \dots \rrbracket \\
&= 3\{(\mathbf{Head} : x; \mathbf{Tail} : g(\mathbf{Succ}(x)))\} + \llbracket \mathbf{Zero} \rrbracket \\
&= 3 \max\{\llbracket x \rrbracket, \llbracket g(\mathbf{Succ}(x)) \rrbracket\} + 1 \\
&= 3 \max\{0, 1\} + 1 \\
&= 4
\end{aligned}$$

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.
2. Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions*. PhD thesis, University of Oslo, 1997.
3. A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel ed., *Proc. of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, 24-30, North Holland, Amsterdam, 1964
4. R. Cockett and R. Seely. Polarized category theory, modules and game semantics. *Theory and Application of Categories*, 18:4–101, 2007.
5. R. Cockett and R. Seely. Finite sum-product logic. *Theory and Applications of Categories*, 8(5):63–99, 2001.
6. Loïc Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991.
7. M. Hofmann. Type systems for polynomial-time computation. Habilitation thesis, University of Darmstadt, 1999.
8. M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1), pages 57-85, 2003
9. D. Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, 325-333, 1993
10. D. Leivant and J.-Y. Marion. Ramified Recurrence and Computational Complexity II: Substitution and Poly-space. In *Proc. CSL'94, Springer LNCS*, 933, pages 486-500, 1994
11. Yiannis N. Moschovakis. On Colson's theorem. Available at <http://www.mathphys.ntua.gr/logic/symposium/articles/moschova.ps>, 1999. Invited talk at 2nd Panhellenic Logic Symposium, Delphi, Greece.
12. Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In *Lecture Notes in Computer Science*, volume 2303, pages 357–371, 2002.
13. V. Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, 2000.

A Syntax

The language is functional in style and the syntax reflects that. The most unique features of the syntax are in the declarations of data types, both inductive and coinductive, and in the use of folds and unfolds for recursion. The Kleene star (*) and plus (+) are metasyntactic symbols to stand for zero-or-more or one-or-more, respectively, occurrences. Ellipses (...) are also used to show a list of zero-or-more of something. All other symbols (notably parentheses, colons and vertical bars) are literal syntactic features.

$Declaration := \mathbf{data} D(A_1, \dots, A_m) \mapsto C = \{Constructor^+\}$	Inductive data type
$Declaration := \mathbf{data} C \mapsto D(A_1, \dots, A_m) = \{Destructor^+\}$	Coinductive data type
$Declaration := f = x_1, \dots, x_m \mid y_1, \dots, y_n. Term$	Function declaration
$Constructor := C : \mid A_1, \dots, A_n \mapsto C$	
$Destructor := D : \mid C, A_1, \dots, A_n \mapsto B$	
$Term := x$	Variable
$Term := (Term, \dots, Term)$	Product
$Term := f(Term, \dots, Term)$	Function application
$Term := \mathbf{C}(Term, \dots, Term)$	Construction
$Term := \mathbf{D}(Term, Term, \dots, Term)$	Destruction
$Term := \mathbf{case} Term \mathbf{of} \{ Branch^+ \}$	Case
$Term := \mathbf{peek} Term \mathbf{of} \{ Branch^+ \}$	Peek
$Term := (Cobranch^+)$	Record
$Term := \mathbf{fold} f(x, y^*) \mathbf{as} \{ Branch^+ \} \mathbf{in} f(Term^+)$	Fold
$Term := \mathbf{unfold} f(x^*) \mathbf{as} (Cobranch^+) \mathbf{in} f(Term^*)$	Unfold
$Branch := \mathbf{C}(x_1, \dots, x_n). Term$	Constructor pattern
$Branch := (x_1, \dots, x_n). Term$	Product projection
$Cobranch := \mathbf{D} : x_1, \dots, x_n. Term$	Destructor

The convention used above is that x and y are used to stand for variable names and A and C stand for type variables.