

# Homework 4: Recursive Types and Polymorphism

15-814: Types and Programming Languages  
TA: Bernardo Toninho (btoninho@cs.cmu.edu)

Out: 25/10/11

Due: 8/11/11

## 1 Programming with Recursive Types

In the previous homework we did a little bit of programming with co-inductive types, in particular with streams of natural numbers. In this exercise, we will move to the more general case of recursive types.

We will consider a language with natural numbers, booleans, function types, products, sums, unit and recursive types (you may use conditional branching, addition and multiplication, mod, equality and inequality testing without defining it). Recall the typing rules and dynamics for fold and unfold:

$$\frac{\Gamma \vdash e : [(\mu t. \tau)/t]\tau}{\Gamma \vdash \text{fold}[t. \tau](e) : \mu t. \tau} \quad \frac{\Gamma \vdash e : \mu t. \tau}{\Gamma \vdash \text{unfold}(e) : [(\mu t. \tau)/t]\tau}$$
$$\frac{}{\text{fold}[t. \tau](e) \text{ val}} \quad \frac{e \mapsto e'}{\text{unfold}(e) \mapsto \text{unfold}(e')} \quad \frac{}{\text{unfold}(\text{fold}[t. \tau](e)) \mapsto e}$$

Also, recall that we may define the general recursive expression  $\text{fix}[\tau](x.e)$ . Where  $x$  stands for the recursive expression itself in  $e$ . Streams of natural numbers are just a special case of recursive types. In particular, the type  $\sigma$  of streams of natural numbers can be defined as:

$$\sigma \triangleq \mu t. \text{nat} \times t$$

and  $\text{hd}$ ,  $\text{tl}$  and  $\text{corec}$  can also be defined through fold and unfold. In the following exercises, you may use these constructs without defining them.

In the previous assignment, you have defined  $\text{map}$  and  $\text{merge}$  functions on streams. Another very useful function is a filter function, that takes a stream, a function from natural numbers to booleans and produces the stream in which the numbers for which the function returns false are filtered out.

**Task 1 (Filtering Streams)** Define a function  $\text{filter} : \sigma \rightarrow (\text{nat} \rightarrow \text{bool}) \rightarrow \sigma$  with the behavior described above.

An interesting stream is the one that generates all the prime numbers. You will now build this stream by defining an auxiliary function that takes the stream of all natural numbers, starting at 2, and computes the stream of all the prime numbers using the sieve of Eratosthenes.

**Task 2 (Primes)** Define a function  $\text{primes} : \sigma \rightarrow \sigma$  that given the sequence of all natural numbers, starting at 2, computes the infinite sequence of all the prime numbers.

(Hint) Recall the Sieve of Eratosthenes:

1. Take all the natural numbers starting at 2.
2. The first element of the above sequence is a prime number. Remove it from the list, as well as all its multiples.
3. Repeat 2 until all the required primes have been generated.

## 2 “Dynamic” Typing

Consider the following dynamically typed language:

$$E ::= x \mid \bar{n} \mid \mathbf{z} \mid \mathbf{succ}(E) \mid \mathbf{ifz}(E)\{\mathbf{z} \Rightarrow E_0 \mid \mathbf{succ}(x) \Rightarrow E_1\} \mid \lambda x.E \\ E_1 E_2 \mid \mathbf{fix} x \mathbf{is} E \mid \langle E_1, E_2 \rangle \mid \mathbf{fst}(E) \mid \mathbf{snd}(E) \mid \mathbf{nil} \mid \mathbf{isnil}(E)$$

The values are the numerals, the lambda abstraction, the empty product  $\mathbf{nil}$  and the pairs  $\langle E_1, E_2 \rangle$ . The built-in function  $\mathbf{isnil}(E)$  returns  $\bar{0}$  if  $E$  evaluates to the empty product and  $\bar{1}$  otherwise.

As we did in class, we can make explicit all the tagging and checking that is required at runtime to obtain (a weak notion of) safety. To this effect, consider a hybrid language like that of Chapter 22 of the PFPL book, but with two more extra classes of data for products and sums ( $\mathbf{prod}$  and  $\mathbf{sum}$ , respectively).

The Ackermann function is defined as follows:

$$A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & n = 0 \\ A(m - 1, A(m, n - 1)) & \text{otherwise} \end{cases}$$

**Task 3 (The Ackermann Function)** Write a function that computes the Ackermann function, using the dynamically typed language given above (you may use subtraction without defining it).

**Task 4** Translate the function from the previous task into the hybrid language, making explicit all necessary tags and checks and not introducing any optimizations (i.e. all variables bound by  $\lambda$  abstractions and the fix point iterator should be of type  $\mathbf{dyn}$ ).

**Task 5** Optimize the function of the previous task by removing all unnecessary casts and news (don't forget to write the appropriate type annotations).

## 3 System F

Recall the statics of System F:

$$\frac{}{\Delta, X \text{ type} \vdash X \text{ type}} \quad \frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}} \quad \frac{\Delta, X \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \forall X. \tau \text{ type}} \\ \frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau} \quad \frac{\Delta; \Gamma, x : \tau_1 \vdash M : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2} \quad \frac{\Delta; \Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad \Delta; \Gamma \vdash N : \tau_2}{\Delta; \Gamma \vdash M N : \tau_2} \\ \frac{\Delta, X \text{ type}; \Gamma \vdash M : \tau}{\Delta; \Gamma \vdash \Lambda X. M : \forall X. \tau} \quad \frac{\Delta \vdash \tau \text{ type} \quad \Delta; \Gamma \vdash M : \forall X. \tau_1}{\Delta; \Gamma \vdash M[\tau] : [\tau/X]\tau_1}$$

As we did in class, we will consider definitional equivalence  $\equiv$  as the least congruence satisfying the following two rules (that is, there are additional rules that specify that  $\equiv$  is reflexive, transitive, symmetric and compatible with all the constructs of the language):

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash M : \tau_2 \quad \Delta; \Gamma \vdash N : \tau_1}{\Delta; \Gamma \vdash (\lambda x : \tau_1. M) N \equiv [N/x]M : \tau_2} \quad \frac{\Delta, t \text{ type}; \Gamma \vdash M : \tau \quad \Delta \vdash \rho \text{ type}}{\Delta; \Gamma \vdash (\Lambda X. M)[\rho] \equiv [\rho/X]M : [\rho/X]\tau}$$

As detailed in the PFPL book, natural numbers can be encoded in System F through the following type and terms:

$$\begin{aligned} \mathbf{nat} &\triangleq \forall X. (X \rightarrow (X \rightarrow X) \rightarrow X) \\ \mathbf{z} &\triangleq \Lambda X. \lambda z : X. \lambda s : X \rightarrow X. z \\ \mathbf{s}(M) &\triangleq \Lambda X. \lambda z : X. \lambda s : X \rightarrow X. M[X](z)(s) \\ \mathbf{rec}(M; M_0; x. M_1) &\triangleq M[\tau](M_0)(\lambda x : \tau. M_1) \end{aligned}$$

We will now consider encoding streams of natural numbers.

To encode streams, it is easier to consider an extension of System F with existential and product types (which are definable in F). Recall the statics of products and existentials:

$$\begin{array}{c}
\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \times \tau_2 \text{ type}} \quad \frac{\Delta, X \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \exists X. \tau \text{ type}} \\
\frac{\Delta; \Gamma \vdash M : \tau_1 \quad \Delta; \Gamma \vdash N : \tau_2}{\Delta; \Gamma \vdash \langle M; N \rangle : \tau_1 \times \tau_2} \quad \frac{\Delta : \Gamma \vdash M : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \pi_i(M) : \tau_i} \\
\frac{\Delta \vdash \rho \text{ type} \quad \Delta, X \text{ type} \vdash \tau \text{ type} \quad \Delta; \Gamma \vdash M : [\rho/X]\tau}{\Delta; \Gamma \vdash \text{pack } \rho \text{ with } M \text{ as } \exists X. \tau : \exists X. \tau} \\
\frac{\Delta; \Gamma \vdash M_1 : \exists X. \tau \quad \Delta, X \text{ type}; \Gamma, x : \tau \vdash M_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta; \Gamma \vdash \text{open } M_1 \text{ as } X \text{ with } x : \tau \text{ in } M_2 : \tau_2}
\end{array}$$

Definitional equivalence for existentials is given below:

$$\frac{\Delta \vdash \rho \text{ type} \quad \Delta, X \text{ type} \vdash \tau \text{ type} \quad \Delta; \Gamma \vdash M_1 : [\rho/X]\tau \quad \Delta, X \text{ type}; \Gamma, x : \tau \vdash M_2 : \tau_2 \quad \Delta \vdash \tau_2 \text{ type}}{\Delta; \Gamma \vdash \text{open } (\text{pack } \rho \text{ with } M_1 \text{ as } \exists X. \tau) \text{ as } X \text{ with } x : \tau \text{ in } M_2 \equiv [\rho, M_1/X, x]M_2 : \tau_2}$$

With this setup, we can view streams in terms of their introduction form (the co-recursor): a stream is composed by a tuple of three objects, a seed or a “current state” element, a function that maps the seed to a natural number (the head) and a function that creates a new seed from the current one (the tail).

**Task 6 (Encoding Streams)** *Define the type stream in System F extended existentials and pairs. Give suitable (according to your encoding) definitions of hd, tl and corec that satisfy the following equivalences:*

- $\text{hd}(\text{corec}(M; x.M_0; x.M_1)) \equiv [M/x]M_0$
- $\text{tl}(\text{corec}(M; x.M_0; x.M_1)) \equiv \text{corec}([M/x]M_1; x.M_0; x.M_1)$

### 3.1 General Inductive and Co-Inductive Types

As was shown in class, the type  $\omega$  of natural numbers can be understood as a particular instance of an initial algebra for a polynomial type operator. Likewise, streams of natural numbers are a particular instance of a final coalgebra for a polynomial type operator. Using the notation of the PFPL book (Chapter 17), we can consider inductive and co-inductive types, defined in terms of type operators using the following syntax of types  $T$  ( $\dots$  denotes sums, products, arrow, unit and void):

$$T ::= X \mid \mu_i X.F(X) \mid \mu_f X.F(X) \mid \dots$$

where  $X$  denotes a type variable,  $\mu_i$  the inductive type defined by a type operator  $X.F(X)$  and  $\mu_f$  the co-inductive type defined by a type operator  $X.F(X)$ . In class we mentioned polynomial type operators, that is, type operators not containing arrow types (It turns out that the idea can be generalized to *positive* type operators, which allow for instances of arrow types but where the type variable cannot occur on the left of the arrow).

For instance, the types  $\omega$  of natural numbers and  $\sigma$  of streams of natural numbers can be defined as:

$$\begin{array}{l}
\omega \triangleq \mu_i X.N(X) \\
\sigma \triangleq \mu_f X.S(X)
\end{array}$$

where  $N(X) = 1 + X$ , satisfying the isomorphism  $\omega \cong N(\omega)$ , that is,  $\omega$  is a fixed point of the type operator  $N(X)$  (in particular, a least fixed point). Similarly,  $S(X) = \omega \times X$ , satisfying

the isomorphism  $\sigma \cong S(\sigma)$ . A more detailed explanation of this language is given in the PFPL book.

All types of the form mentioned above are present in System F. Given a polynomial type operator  $X.F(X)$ , we can encode inductive and coinductive types as follows:

$$\begin{aligned}\mu_i X.F(X) &\triangleq \forall X.(F(X) \rightarrow X) \rightarrow X \\ \mu_f X.F(X) &\triangleq \exists X.(X \rightarrow F(X)) \times X\end{aligned}$$

To ensure that this is correct, we need to show that the types given above have the required algebraic properties. An inductive type can be understood as an initial algebra for a given polynomial type operator. To review, an algebra of a type operator  $F$  is a pairing of a type  $X$  and a function  $f : F(X) \rightarrow X$ . For an algebra  $(X, f)$  to be initial, it must be the case that for any other algebra  $(X', f')$  there must exist a unique function  $h : X \rightarrow X'$  such that the following diagram commutes:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(h)} & F(X') \\ f \downarrow & & \downarrow f' \\ X & \xrightarrow{h} & X' \end{array}$$

**Task 7 (Inductive Types)** Consider the encoding of inductive types given above (we shall refer to it as  $T_i$ ). Define a pair of functions  $\text{fold} : \forall X.(F(X) \rightarrow X) \rightarrow T_i \rightarrow X$  and  $\text{in} : F(T_i) \rightarrow T_i$  such that the following diagram commutes:

$$\begin{array}{ccc} F(T_i) & \xrightarrow{F(\text{fold}[X](f))} & F(X) \\ \text{in} \downarrow & & \downarrow f \\ T_i & \xrightarrow{\text{fold}[X](f)} & X \end{array}$$

**(Hint)** You will need to use the action of the type operator  $F$  when defining  $\text{in}$ . You may do this by just writing  $F(M)$  for some term  $M$ , producing the appropriate action. Also note that at the level of types  $F(A \rightarrow B) = F(A) \rightarrow F(B)$ .

Similarly, a coinductive type can be understood as a final coalgebra of a type operator  $F$ . A coalgebra consists of a pair  $(X, f)$  where  $X$  is a type and  $f : X \rightarrow F(X)$ . For such a coalgebra to be final, there must exist a unique function  $h : X' \rightarrow X$  such that the following diagram commutes:

$$\begin{array}{ccc} X' & \xrightarrow{h} & X \\ f' \downarrow & & \downarrow f \\ F(X') & \xrightarrow{F(h)} & F(X) \end{array}$$

**Task 8 (Coinductive Types)** Consider the encoding of coinductive types given above, referred to here as  $T_f$ . Define a pair of functions  $\text{unfold} : \forall X.(X \rightarrow F(X)) \rightarrow X \rightarrow T_f$  and  $\text{out} : T_f \rightarrow F(T_f)$  such that the following diagram commutes:

$$\begin{array}{ccc}
X & \xrightarrow{\text{unfold}[X](f)} & T_f \\
\downarrow f & & \downarrow \text{out} \\
F(X) & \xrightarrow{F(\text{unfold}[X](f))} & F(T_f)
\end{array}$$

In the two tasks given above, you defined functions `fold` and `unfold` that partially fulfill the requirements of an initial (resp. final) algebra (resp. coalgebra). Technically, the functions witness the *weak* initiality and finality of the given algebras. The uniqueness requirement will follow from parametricity, but we will not discuss that here.

**Task 9 (Streams yet again)** Recall that streams are defined by the following coinductive type  $\mu_f X.\omega \times X$ . Using the encodings given above, write out the translation of this coinductive type in System F. Show that the type you obtain is isomorphic to the one you defined in Task 6. You should also observe that `hd`, `tl` and `corec` can be directly obtained from `unfold` and `out`.