Homework 5: **PCF**, **FPC** and **MA**

15-814: Types and Programming Languages
Fall 2017
Instructor: Karl Crary
TA: Yong Kiam Tan

Out: Oct 30, 2017 (07 pm)
Due: Nov 13, 2017 (11 pm)

Notes:

- Welcome to 15-814's fifth homework assignment!
- Please email your work as a PDF file to `yongkiat@cs.cmu.edu` titled "15-814 Homework 5". Your PDF should be named "<your-name>-hw5-sol.pdf".

1 Halting Problem in PCF

Recall the language **PCF**:

$$\begin{array}{rcl} \tau & ::= & \texttt{nat} \mid \tau \to \tau \\ e & ::= & x \mid \texttt{z} \mid \texttt{s}(e) \mid \texttt{ifz}(e; e; x.e) \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{fix}\ x : \tau.e \end{array}$$

Consider a term $H : (\texttt{nat} \to \texttt{nat}) \to \texttt{nat}$ with the following properties:

1. For all $f : \texttt{nat} \to \texttt{nat}$, either $H\ f \mapsto^* \texttt{z}$ or $H\ f \mapsto^* \texttt{s}(\texttt{z})$ (i.e. H always terminates and evaluates to either $\overline{0}$ or $\overline{1}$.)
2. $H\ f \mapsto^* \texttt{z}$ iff there exists n such that $f\ \texttt{z} \mapsto^* \overline{n}$ (i.e. $f\ \texttt{z}$ converges to a value.)
3. $H\ f \mapsto^* \texttt{s}(\texttt{z})$ iff $f\ \texttt{z}$ diverges.

Task 1 *Prove that H is not definable in* **PCF**.

(Hint) *Suppose H exists. Define a term D (which may refer to H) such that D diverges iff $H\ D \mapsto^* \texttt{z}$ (to make a term diverge, you can easily write an infinite loop.) Then consider to what $H\ D$ should evaluate.*

This is a weaker version of the famous result that the Halting Problem is undecidable (in a sufficiently powerful language). In the general statement, H accepts a representation of the code of a function f instead of the function itself. This problem is also undecidable for **PCF**.

To make precise the idea of having H accept a representation of the code of a function, we use a technique called *Gödel-numbering*, which assigns a unique natural number to α-equivalence classes of terms. We will not go into details of how such a representation is computed, but you can see PFPL 9.4 for more information. We will write $\ulcorner e \urcorner$ for the Gödel number of an expression e. Since natural numbers are available in **PCF**, this gives us a way of passing around representations of expressions as values that can be inspected arbitrarily (as opposed to functions themselves, which can only be "inspected" by application.)

We will also generalize the definition of H so that it accepts a natural number input as well as the function. We will call this generalization H'.

$H' : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$ has the following properties.

1. For all $f : \texttt{nat} \to \texttt{nat}$ and $n : \texttt{nat}$, either $H' \ulcorner f \urcorner\, n \mapsto^* \texttt{z}$ or $H' \ulcorner f \urcorner\, n \mapsto^* \texttt{s}(\texttt{z})$ (i.e. H' always terminates and evaluates to either $\overline{0}$ or $\overline{1}$.)
2. $H' \ulcorner f \urcorner\, n \mapsto^* \texttt{z}$ iff there exists m such that $f\ n \mapsto^* \overline{m}$ (i.e. $f\ n$ converges to a value.)
3. $H' \ulcorner f \urcorner\, n \mapsto^* \texttt{s}(\texttt{z})$ iff $f\ n$ diverges.

Task 2 *Prove that H' is not definable in* **PCF**.

2 Defining Streams

In this section, we will define and manipulate infinite streams of natural numbers using recursive types. We will be working in **FPC** extended with general recursion. As we saw in class, $\texttt{fix}\ x : \tau.e$ can be encoded using recursive types, so this is just for convenience.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \mu\alpha.\tau \\ e & ::= & \cdots \mid \texttt{fold}[\alpha.\tau](e) \mid \texttt{unfold}(e) \mid \texttt{fix}\ x : \tau.e \end{array}$$

Note that there is a type annotation $[\alpha.\tau]$ on $\texttt{fold}$. This follows the presentation in PFPL 20, and is slightly different from the presentation we saw in class. The annotation is used in the typing judgment to indicate the recursive type:

$$\frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau}{\Gamma \vdash \texttt{fold}[\alpha.\tau](e) : \mu\alpha.\tau}\ (\textsc{Fold})$$

We will define the stream type in **FPC** as follows[1]:

$$\texttt{stream} \triangleq \mu\alpha.\texttt{unit} \to \texttt{nat} \times \alpha$$

$$\texttt{hd}(e) \triangleq \pi_1(\texttt{unfold}(e)\ \star)$$

$$\texttt{tl}(e) \triangleq \pi_2(\texttt{unfold}(e)\ \star)$$

For example, we can define the constant stream $\texttt{ones} \triangleq 1, 1, \cdots$ as follows:

$$\texttt{ones} = \texttt{fix}\ x : \texttt{stream}.\underbrace{\texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\overbrace{\lambda\_{:}\texttt{unit}.\langle 1, x\rangle}^{\texttt{unit}\to\texttt{nat}\times\texttt{stream}})}_{\texttt{stream}}$$

We can check that it has the correct behavior:

$$\texttt{hd}(\texttt{ones}) \triangleq \pi_1(\texttt{unfold}(\texttt{ones})\ \star)$$

[1]For those interested, its encoding as a coinductive type is $\texttt{stream} \triangleq \nu\alpha.\texttt{nat} \times \alpha$.

$$\begin{aligned}
&\mapsto \pi_1(\texttt{unfold}(\texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle 1, \texttt{ones}\rangle))\ \star)\\
&\mapsto \pi_1(\lambda\_{:}\texttt{unit}.\langle 1, \texttt{ones}\rangle\ \star)\\
&\mapsto \pi_1(\langle 1, \texttt{ones}\rangle)\\
&\mapsto \pi_1(\langle 1, \texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle 1, \texttt{ones}\rangle)\rangle)\\
&\mapsto 1
\end{aligned}$$

Notice that our `fix` is eager, so `ones` is eagerly evaluated in the second last evaluation step. A similar derivation will show that `tl(ones)` essentially steps to itself (modulo eager evaluation):

$$\texttt{tl}(\texttt{ones}) \mapsto^* \texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle 1, \texttt{ones}\rangle)$$

For each of the following tasks, you should **briefly** explain the intuition behind your answer. You **must include the appropriate type annotation** for `fold` whenever it is used in your answer.

(Hint) (Optional) You may also find it helpful to annotate other parts of your code with their types as we did in the definition of `ones`.

Task 3 *Define the function* $\texttt{fromLoop} : (\alpha \to \alpha \times \texttt{nat}) \to \alpha \to \texttt{stream}$*, which takes a value* v *of type* α *and a function* f *of type* $\alpha \to \alpha \times \texttt{nat}$*, successively applies* f *to* v *to get values of type* `nat`*, and constructs a stream from these natural numbers.*

Task 4 *Use* `fromLoop` *to construct the following two streams.*

1. *Given a natural number* k*, a stream of natural numbers starting from* k*.*
2. *The stream of natural numbers.*

Task 5 *Define the function,* $\texttt{map} : (\texttt{nat} \to \texttt{nat}) \to \texttt{stream} \to \texttt{stream}$*, which takes a function* f *and stream* s *and applies* f *to every element in the stream* s*.*

Task 6 *Define the function* $\texttt{streamfix} : (\texttt{stream} \to \texttt{stream}) \to \texttt{stream}$*, which takes a function* f *and applies that successively to obtain a stream. (Carefully define this function considering that we are working in the eager, call-by-value version of* **FPC***.)*

Task 7 *Note that the stream of natural numbers has the special property that it can be obtained by adding* 1 *to every element in the stream and then prepending* 0 *to the result. Use this property to define the stream of natural numbers using* `map` *and* `streamfix`*.*

Task 8 *What would happen if you use* `streamfix` *with the identity function?*

3 Monadization

Consider the following two languages, **L1** and **L2**. Both languages extend **PCF** with primitives for input and output. **L2** maintains a separation between expressions e and commands m, while **L1** removes this distinction.

The syntax of **L1** is as follows:

$$\begin{array}{rcl} \tau & ::= & \texttt{nat} \mid \tau \to \tau \mid \tau \times \tau \\ e & ::= & x \mid \overline{n} \mid \texttt{ifz}(e; e; x.e) \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{fix}\ x : \tau.e \mid \langle e, e\rangle \mid \pi_1 e \mid \pi_2 e \mid \texttt{input} \mid \texttt{output}(e) \end{array}$$

The syntax of **L2** is as follows:

$$\begin{array}{rcl} \tau & ::= & \texttt{nat} \mid \tau \to \tau \mid \tau \times \tau \mid \tau\ \texttt{cmd} \\ e & ::= & x \mid \overline{n} \mid \texttt{ifz}(e; e; x.e) \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{fix}\ x : \tau.e \mid \langle e, e\rangle \mid \pi_1 e \mid \pi_2 e \mid \texttt{cmd}(m) \\ m & ::= & \texttt{ret}(e) \mid \texttt{bnd}(m; x.m) \mid \texttt{force}(e) \mid \texttt{input} \mid \texttt{output}(e) \end{array}$$

We define a program transformation written $\bar{\cdot}$. For any **L1** expression e, the translation $\overline{e}$ should be an **L2** command. To define this transformation for expressions, we will have to first define it for types and contexts, such that if $\Gamma \vdash e : \tau$, then $\overline{\Gamma} \vdash \overline{e} \mathrel{\dot\sim} \overline{\tau}$. We will define the type and context transformations for you.

$$\begin{array}{rcl} \overline{\texttt{nat}} & = & \texttt{nat} \\ \overline{\tau_1 \to \tau_2} & = & \overline{\tau_1} \to \overline{\tau_2}\ \texttt{cmd} \\ \overline{\tau_1 \times \tau_2} & = & \overline{\tau_1} \times \overline{\tau_2} \\ \overline{\Gamma, x : \tau} & = & \overline{\Gamma}, x : \overline{\tau} \\ \bar{\cdot} & = & \cdot \end{array}$$

The last statement implies that the empty context in **L1** transforms to the empty context in **L2**.

You should assume the usual typing rules for all the constructs. The typing rules for `input` and `output`(e) are given below. Intuitively, `input` models reading a natural number from the user, while `output`(e) writes a natural number e, and then returns a success code.

$$\frac{}{\Gamma \vdash \texttt{input} : \texttt{nat}}\ (\text{L1-Input}) \qquad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{output}(e) : \texttt{nat}}\ (\text{L1-Output})$$

$$\frac{}{\Gamma \vdash \texttt{input} \mathrel{\dot\sim} \texttt{nat}}\ (\text{L2-Input}) \qquad \frac{\Gamma \vdash e : \texttt{nat}}{\Gamma \vdash \texttt{output}(e) \mathrel{\dot\sim} \texttt{nat}}\ (\text{L2-Output})$$

Task 9 *Define $\overline{e}$ inductively for each expression e in* **L1**.