Homework 5: PCF, FPC and MA
15-814: Types and Programming Languages
Fall 2016
Instructor: Karl Crary
TA: Ankush Das

SOLUTIONS

Out: Nov 02, 2016 (03 pm)
Due: Nov 16, 2016 (11 pm)

Notes:

• Welcome to 15-814’s fifth homework assignment!
• Please email your work as a PDF file to ankushd@cs.cmu.edu titled “15-814 Homework 5”. Your PDF should be named “<your-name>-hw5-sol.pdf”.
• Note that if you’ve not attempted a problem in the first submission, you will not be allowed to attempt it in the resubmission.

1 Halting Problem in PCF

Recall the language PCF:

\[
\begin{align*}
\tau &::= \text{nat} | \tau \rightarrow \tau \\
\mathit{e} &::= x | z | \mathit{s}(\mathit{e}) | \mathit{ifz}(\mathit{e};\mathit{e};x.e) | \lambda x . \tau . e | e \ e | \mathit{fix}\{\tau\}(x.e) \\
\end{align*}
\]

Consider a term \( H : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \) with the following properties:

1. For all \( f : \text{nat} \rightarrow \text{nat} \), either \( H f \rightarrow^* z \) or \( H f \rightarrow^* s(z) \) (i.e. \( H \) always terminates and evaluates to either \( 0 \) or \( 1 \).)
2. \( H f \rightarrow^* z \) iff there exists \( n \) such that \( f z \rightarrow^* n \) (i.e. \( f z \) converges to a value.)
3. \( H f \rightarrow^* s(z) \) iff \( f z \) 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 \( H D \rightarrow^* z \) (to make a term diverge, you can easily write an infinite loop.) Then consider to what \( H D \) should evaluate.

Solution: Define \( \mathit{loop} \triangleq \mathit{fix}\{\text{nat}\}(x.x) \). It is easy to see that \( \mathit{loop} \) diverges.

Let \( D \triangleq \mathit{fix}\{\text{nat} \rightarrow \text{nat}\}(d.\lambda \d. \text{nat.}\mathit{ifz}(H d; \mathit{loop}; \_z)) \).

By property 1, either \( H D \rightarrow^* z \) or \( H D \rightarrow^* s(z) \). If \( H D \rightarrow^* z \), then evaluating \( D z \), we see that

\[
D z \rightarrow^* \mathit{ifz}(H D; \mathit{loop}; \_z) \rightarrow^* \mathit{ifz}(z; \mathit{loop}; \_z) \rightarrow \mathit{loop}
\]
and thus \(Dz\) diverges, but this contradicts property 2.

If \(HD\mapsto^*s(z)\), then evaluating \(Dz\), we see that

\[
Dz \mapsto^* \text{ifz}(HD;\text{loop};z) \mapsto^* \text{ifz}(s(z);\text{loop};z) \mapsto z
\]

so \(Dz\) converges, but this contradicts property 3.

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 \textbf{PCF}.

To make precise the idea of having \(H\) accept a representation of the code of a function, we use a technique called \textit{Gödel-numbering}, which assigns a unique natural number to \(\alpha\)-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 \(\llbracket e \rrbracket\) for the Gödel number of an expression \(e\). Since natural numbers are available in \textbf{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 \(H'\).

\(H' : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}\) has the following properties.

1. For all \(f : \text{nat} \rightarrow \text{nat}\) and \(n : \text{nat}\), either \(H' \mapsto f^\uparrow n \mapsto^* z\) or \(H' \mapsto f^\uparrow n \mapsto^* s(z)\) (i.e. \(H'\) always terminates and evaluates to either 0 or 1.)

2. \(H' \mapsto f^\uparrow n \mapsto^* z\) iff there exists \(m\) such that \(f^\uparrow n \mapsto^* m\) (i.e. \(f^\uparrow n\) converges to a value.)

3. \(H' \mapsto f^\uparrow n \mapsto^* s(z)\) iff \(f^\uparrow n\) diverges.

**Task 2** Prove that \(H'\) is not definable in \textbf{PCF}.

**Solution:** Suppose \(H'\) exists. Then let

\[
d \triangleq \lambda n : \text{nat}. \text{ifz}(H' n n; \text{loop}; z)
\]

By assumption, either \(H' \mapsto d^\uparrow \mapsto^* z\) or \(H' \mapsto d^\uparrow \mapsto^* s(z)\). In the first case, evaluating \(d^\uparrow d^\uparrow\) gives

\[
d^\uparrow d^\uparrow \mapsto^* \text{ifz}(H' d^\uparrow d^\uparrow; \text{loop}; z) \mapsto^* \text{ifz}(z; \text{loop}; z) \mapsto^* \text{loop}
\]

so \(d^\uparrow d^\uparrow\) diverges, but the definition of \(H'\) states that it should converge.

Similarly, in the second case, evaluating \(d^\uparrow d^\uparrow\) gives

\[
d^\uparrow d^\uparrow \mapsto^* \text{ifz}(H' d^\uparrow d^\uparrow; \text{loop}; z) \mapsto^* \text{ifz}(s(z); \text{loop}; z) \mapsto^* z
\]

so \(d^\uparrow d^\uparrow\) converges, but the definition of \(H'\) states that it should diverge. In either case, we have a contradiction.
2 Defining Coinductive Types

In this section, we will encode coinductive types as recursive types in FPC. We will be working in FPC extended with general recursion (\(\text{fix}\{\tau\}(x.e)\) can be encoded using recursive types, so this is just for convenience).

\[
\begin{align*}
\tau \ ::= & \ t \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \text{rec}(t, \tau) \\
e \ ::= & \ \cdots \mid \text{roll}\{t, \tau\}(e) \mid \text{unroll}\{t, \tau\}(e) \mid \text{fix}\{\tau\}(x.e)
\end{align*}
\]

Let’s consider the special case of streams for now. A \(\tau\) stream is an infinite stream of elements of type \(\tau\). A stream \(e\) is characterized by its head \(\text{hd}(e)\) and tail \(\text{tl}(e)\). We consider streams of natural numbers here. For those interested, its encoding as a coinductive type is the following.

\[
\text{stream} \triangleq \nu t. \text{nat} \times t
\]

We will define the stream type in FPC as follows:

\[
\text{stream} \triangleq \text{rec}(t.\text{unit} \rightarrow \text{nat} \times t)
\]

\[
\begin{align*}
\text{hd}(e) & \triangleq (\text{unroll}\{t.\text{unit} \rightarrow \text{nat} \times t\}(e) \langle\rangle) \cdot 1 \\
\text{tl}(e) & \triangleq (\text{unroll}\{t.\text{unit} \rightarrow \text{nat} \times t\}(e) \langle\rangle) \cdot r
\end{align*}
\]

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

**Solution:**

\[
\text{helper}(g, f) \triangleq \lambda x : \alpha. \text{roll}\{t.\text{unit} \rightarrow \text{nat} \times t\}(\lambda u : \text{unit}. (\langle f x \rangle \cdot r, g ((\langle f x \rangle \cdot 1))))
\]

\[
\text{fromLoop} \triangleq \lambda f : \alpha \rightarrow \alpha \times \text{nat}. \lambda v : \alpha. (\text{fix}\{\alpha \rightarrow \text{stream}\}(g.\text{helper}(g, f)) v)
\]

**Task 4** Use \(\text{fromLoop}\) 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.

**Solution:**

1. \(\text{gen} = \lambda k : \text{nat}. \text{fromLoop} (\lambda x : \text{nat}. (s(x), x)) k\)
2. \(\text{gen} \ 0\)

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

\[ \text{helper}(g, f) = \lambda s: \text{stream}\cdot \text{roll}\{(t.\text{unit} \rightarrow \text{nat} \times t)\{(\lambda u: \text{unit}. (f \text{hd}(s), g \text{tl}(s))\} \} \]

\[
\text{map} = \lambda f: \text{nat} \rightarrow \text{nat}. \text{fix} \{(\text{stream} \rightarrow \text{stream})\{(g. \text{helper}(g, f)) \}
\]

Task 6 Define the function \( \text{streamfix} : (\text{stream} \rightarrow \text{stream}) \rightarrow \text{stream} \), 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.)

Solution: Let \( \tau = \text{t.unit} + \text{nat} \times \text{t} \).

\[
\text{streamfix} \triangleq \lambda f: \text{stream} \rightarrow \text{stream}. \text{fix} \{(\text{stream} \rightarrow \text{stream})\{(s, \text{roll}\{\tau\}\{(\text{unroll}\{\tau\}\{(f \text{s})\}) \}
\]

Task 7 Note that the list 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 list of natural numbers using \( \text{map} \) and \( \text{streamfix} \).

Solution:

\[
\text{prepend} \triangleq \lambda s: \text{stream}. \text{roll}\{\tau\}\{(\text{unroll}\{\tau\}\{(f \text{s})\}) \}
\]

\[
\text{suc} \triangleq \lambda s: \text{stream}. \text{map}\{(\lambda x: \text{nat}. \text{s}(x))\} s
\]

\[
s \triangleq \text{streamfix}(\text{prepend} \circ \text{suc})
\]

Task 8 What would happen if you use \( \text{streamfix} \) with the identity function?

Solution: It would return a stream that diverges whenever evaluated.

3 Monadization

Consider two languages \( \mathbf{L_1} \) and \( \mathbf{L_2} \). \( \mathbf{L_1} \) and \( \mathbf{L_2} \) are pure and impure extensions of PCF respectively. The syntax of \( \mathbf{L_1} \) is as follows.

\[
\tau ::= \text{nat} | \tau \rightarrow \tau | \tau \times \tau
\]

\[
e ::= x | \pi | \text{ifz}(e; e; x.e) | \lambda x: \tau. e | e \cdot e | e \cdot 1 | e \cdot r | \text{input} | \text{output}(e)
\]

The syntax of \( \mathbf{L_2} \) is as follows.

\[
\tau ::= \text{nat} | \tau \rightarrow \tau | \tau \times \tau | \text{cmd}(\tau)
\]

\[
e ::= x | \pi | \text{ifz}(e; e; x.e) | \lambda x: \tau. e | e \cdot e | e \cdot 1 | e \cdot r | \text{cmd}(m)
\]

\[
m ::= \text{ret}(e) | \text{bnd}(e; m) | \text{input} | \text{output}(e)
\]

We define a program transformation written \( \cdot \). For any \( \mathbf{L_1} \) expression \( e \), the translation \( \overline{e} \) should be an \( \mathbf{L_2} \) 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 \( \Gamma \vdash \overline{e} \vdash \overline{\tau} \). We will define the type and context transformations for you.

\[
\overline{\text{nat}} = \text{nat}
\]

\[
\overline{\tau_1 \rightarrow \tau_2} = \overline{\tau_1} \rightarrow \text{cmd}(\overline{\tau_2})
\]

\[
\overline{\tau_1 \times \tau_2} = \overline{\tau_1} \times \overline{\tau_2}
\]

\[
\overline{\Gamma, x : \tau} = \overline{\Gamma}, x : \overline{\tau}
\]

\[
\vdash = \cdot
\]
The last statement implies that the empty context in \( L_1 \) transforms to the empty context in \( L_2 \).

**Task 9** Define \( \tau \) inductively for each expression \( e \) in \( L_1 \).

**Solution:**

\[
\begin{align*}
\overline{\pi} &= \text{ret}(x) \\
\overline{\pi} &= \text{ret}(\pi) \\
\text{ifz}(e; e_0; x.e_1) &= \text{bnd(cmd}(\overline{\tau}); a.\text{bnd}(\text{ifz}(a; \text{cmd}(\overline{\tau})); x.\text{cmd}(\overline{\tau})); b.\text{ret}(b))) \\
\lambda x:\tau.e &= \text{ret}(\lambda x:\tau.\text{cmd}(\overline{\tau})) \\
\overline{e_1 \ e_2} &= \text{bnd(cmd}(\overline{\tau}); x_1.\text{bnd(cmd}(\overline{\tau}_2)); x_2.\text{bnd}(x_1 \ x_2; y.\text{ret}(y)))) \\
\text{fix}\{\tau\}(x.e) &= \text{bnd(cmd}(\overline{\tau}); x.\text{cmd}(\overline{\tau}); y.\text{ret}(y)) \\
\langle e_1, e_2 \rangle &= \text{bnd(cmd}(\overline{\tau}); x_1.\text{bnd(cmd}(\overline{\tau}_2)); x_2.\text{ret}(\langle x_1, x_2 \rangle))) \\
\overline{e \cdot l} &= \text{bnd(cmd}(\overline{\tau}); x.\text{ret}(x \cdot l)) \\
\overline{\overline{\pi} \cdot \overline{r}} &= \text{bnd(cmd}(\overline{\tau}); x.\text{ret}(x \cdot r)) \\
\text{input} &= \text{input} \\
\text{output}(e) &= \text{bnd(cmd}(\overline{\tau}); x.\text{output}(x))
\end{align*}
\]