15-814 Homework hw5

December 1, 2017

1 Halting Problem in PCF

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

Solution: Suppose $H : (\texttt{nat} \to \texttt{nat}) \to \texttt{nat}$ is definable. Consider the term:

$$D = \texttt{fix}\ f : \texttt{nat} \to \texttt{nat}.\lambda\_{:}\texttt{nat}.\texttt{ifz}(H\ f; \Omega; x.\texttt{z})$$

where $\Omega = \texttt{fix}\ x : \texttt{nat}.x$, $\vdash \Omega : \texttt{nat}$, and Ω diverges.

We have:

$$\begin{aligned} D\ \texttt{z} &\mapsto (\lambda\_{:}\texttt{nat}.\texttt{ifz}(H\ D; \Omega; x.\texttt{z}))\ \texttt{z} \\ &\mapsto \texttt{ifz}(H\ D; \Omega; x.\texttt{z}) \end{aligned}$$

By assumption, we know that either $H\ D \mapsto^* \texttt{z}$ or $H\ D \mapsto^* \texttt{s}(\texttt{z})$.

In the first case, $D\ \texttt{z}$ converges by assumption. However, we also take the first branch above, so $D\ \texttt{z} \mapsto^* \Omega$, and so $D\ \texttt{z}$ diverges. By determinism $D\ \texttt{z}$ cannot both diverge and converge, and so this case leads to a contradiction.

In the latter case, $D\ \texttt{z}$ diverges by assumption. Here, we take the second branch, so $D\ \texttt{z} \mapsto^* \texttt{z}$, so $D\ \texttt{z}$ converges. This again leads to a contradiction with determinism of the language.

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

Solution: Suppose that $H' : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$ is definable. Consider the term:

$$D' = \lambda x{:}\texttt{nat}.\texttt{ifz}(H'\ x\ x; \Omega; y.\texttt{z})$$

where Ω diverges, and $D' : \texttt{nat} \to \texttt{nat}$. We have $D' \ulcorner D' \urcorner \mapsto \texttt{ifz}(H' \ulcorner D' \urcorner \ulcorner D' \urcorner; \Omega; y.\texttt{z})$.

Now, by assumption, either $H' \ulcorner D' \urcorner \ulcorner D' \urcorner \mapsto^* \texttt{z}$ or $H' \ulcorner D' \urcorner \ulcorner D' \urcorner \mapsto^* \texttt{s}(\texttt{z})$. In the first case, we have that $D' \ulcorner D' \urcorner$ converges, but this is a contradiction since $D' \ulcorner D' \urcorner \mapsto^* \texttt{ifz}(\texttt{z}; \Omega; y.z) \mapsto \Omega$, which diverges. In the second case, we have that $D' \ulcorner D' \urcorner$ diverges, but this is again a contradiction since $D' \ulcorner D' \urcorner \mapsto^* \texttt{ifz}(\texttt{s}(z); \Omega; y.z) \mapsto \texttt{z}$, which converges.

2 Defining Streams

I will use ML code to illustrate these answers. The datatype and selector definitions for streams are given below (modulo using ML integers instead of natural numbers):

```
datatype stream = Cons of unit -> int * stream
fun hd (Cons f) = #1 (f ())
fun tl (Cons f) = #2 (f ())
```

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* $\texttt{nat}$, *and constructs a stream from these natural numbers.*

Solution: In ML:

```
fun fromLoop f v = Cons(fn ()=> let val (l,r) = f v in (r,fromLoop f l) end);
```

Translating this into **FPC**, writing $t = (\alpha \to \alpha \times \texttt{nat}) \to \alpha \to \texttt{stream}$:

$$\texttt{fromLoop} = \texttt{fix}\ x : t.\lambda f{:}\alpha \to \alpha \times \texttt{nat}.\lambda v{:}\alpha.\texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle \pi_2(f\ v), x\ f\ (\pi_1(f\ v))\rangle)$$

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.*

Solution: In ML:

```
fun natstrk k = fromLoop (fn v => (v+1,v)) k;
val natstr = natstrk 0;
```

Behavior:

```
> hd natstr
val it = 0: int
> hd (tl (tl (tl natstr)))
val it = 3: int
```

Translating:

$$\texttt{natstrk} = \lambda k{:}\texttt{nat}.\texttt{fromLoop}\ (\lambda v{:}\texttt{nat}.\langle(\texttt{s}(v), v)\rangle)\ k$$

$$\texttt{natstr} = \texttt{natstrk}\ \texttt{z}$$

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.

Solution: In ML:

```
fun map f s = Cons(fn () => (f (hd s),map f (tl s)));
```

Translating, writing $t = (\texttt{nat} \to \texttt{nat}) \to \texttt{stream} \to \texttt{stream}$:

$$\texttt{map} = \texttt{fix}\ x : t.\lambda f{:}\texttt{nat} \to \texttt{nat}.\lambda s{:}\texttt{stream}.\texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle f\ (\texttt{hd}\ s), x\ f\ (\texttt{tl}\ s)\rangle)$$

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**.*)*

Solution: The fixpoint equation can just be written in ML, and this would diverge when given f:

```
fun streamfixbad f = f (streamfixbad f);
```

Instead, we must manually "unroll" the stream once:

```
fun streamfix f =
  Cons (fn () => (hd (f (streamfix f)),tl (f (streamfix f))))
```

Translating, writing $t = (\texttt{stream} \to \texttt{stream}) \to \texttt{stream}$:

$\texttt{streamfix} = \texttt{fix}\, x : t.\lambda f{:}\texttt{stream} \to \texttt{stream}.\texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle \texttt{hd}\ (f\ (x\ f)), \texttt{tl}\ (f\ (x\ f))\rangle)$

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`*.*

Solution: In ML:

```
val natstr = streamfix (fn s => Cons (fn _ => (0,map (fn n=>n+1) s) ));
```

Behavior:

```
> hd natstr
val it = 0: int
> hd (tl (tl (tl natstr)))
val it = 3: int
```

Translating:

$\texttt{suc} = \lambda n{:}\texttt{nat}.\texttt{s}(n)$

$\texttt{natstr2} = \texttt{streamfix}\ (\lambda s{:}\texttt{stream}.\texttt{fold}[\alpha.\texttt{unit} \to \texttt{nat} \times \alpha](\lambda\_{:}\texttt{unit}.\langle \texttt{z}, \texttt{map}\ \texttt{suc}\ s\rangle))$

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

Solution: It returns a stream but the stream diverges when it is forced, e.g. when `hd` or `tl` is called on it.

```
> val foo = streamfix (fn s => s)
val foo = Cons fn: stream
hd foo
(* diverges *)
tl foo
(* diverges *)
```

3 Monadization

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

Solution: The translation is mostly straightforward if we follow the types.

$$\overline{\texttt{input}} = \texttt{input} \qquad \overline{\texttt{output}(e)} = \texttt{bnd}(\overline{e}; y.\texttt{output}(y))$$

$$\overline{x} = \texttt{ret}(x) \qquad \overline{\overline{n}} = \texttt{ret}(\overline{n}) \qquad \overline{\lambda x{:}\tau.e} = \texttt{ret}(\lambda x{:}\overline{\tau}.\texttt{cmd}(\overline{e}))$$

$$\overline{\texttt{ifz}(e_1; e_2; x.e_3)} = \texttt{bnd}(\overline{e_1}; x_1.\texttt{force}(\texttt{ifz}(x_1; \texttt{cmd}(\overline{e_2}); x.\texttt{cmd}(\overline{e_3}))))$$

$$\overline{e_1\ e_2} = \texttt{bnd}(\overline{e_1}; x_1.\texttt{bnd}(\overline{e_2}; x_2.\texttt{force}(x_1\ x_2)))$$

We assume that products are eager:

$$\overline{\langle e_1, e_2\rangle} = \texttt{bnd}(\overline{e_1}; x_1.\texttt{bnd}(\overline{e_2}; x_2.\texttt{ret}(\langle x_1, x_2\rangle)))$$

$$\overline{\pi_1 e} = \texttt{bnd}(\overline{e}; x.\texttt{ret}(\pi_1 x)) \qquad \overline{\pi_2 e} = \texttt{bnd}(\overline{e}; x.\texttt{ret}(\pi_2 x))$$

Note: I am not sure if there is an easy solution for the `fix` case that does not diverge. I accepted all solutions that are well-typed.

I got the following solution from students in class, I think it has the right behavior:

$$\overline{\texttt{fix}\ x : \tau.e} = \texttt{force}(\texttt{fix}\ f : \overline{\tau}\ \texttt{cmd}.\texttt{cmd}([\texttt{force}(f)/\texttt{ret}(x)]\overline{e}))$$

My original solution diverges:

$$\overline{\texttt{fix}\ x : \tau.e} = \texttt{force}(\texttt{fix}\ f : \overline{\tau}\ \texttt{cmd}.\texttt{cmd}(\texttt{bnd}(\texttt{force}(f); x.\overline{e})))$$