1 Halting Problem in PCF

Task 1 Prove that \(H\) is not definable in PCF.

Solution: Suppose \(H : (\text{nat} \to \text{nat}) \to \text{nat}\) is definable. Consider the term:
\[
D = \text{fix } f : \text{nat} \to \text{nat}. \lambda x : \text{nat} . \text{ifz}(H f ; \Omega ; x . z)
\]
where \(\Omega = \text{fix } x : \text{nat}, \uplus \Omega : \text{nat}\), and \(\Omega\) diverges.

We have:
\[
D z \mapsto (\lambda x : \text{nat} . \text{ifz}(H D ; \Omega ; x . z)) z \mapsto \text{ifz}(H D ; \Omega ; x . z)
\]

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

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

In the latter case, \(D z\) diverges by assumption. Here, we take the second branch, so \(D z \mapsto^* z\), so \(D z\) 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 \(H' : \text{nat} \to \text{nat} \to \text{nat}\) is definable. Consider the term:
\[
D' = \lambda x : \text{nat} . \text{ifz}(H' x z ; \Omega ; y . z)
\]
where \(\Omega\) diverges, and \(D' : \text{nat} \to \text{nat}\). We have \(D' D' \mapsto^* \text{ifz}(H' D' ; \Omega ; y . z)\).

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

```ml
datatype stream = Cons of unit -> int * stream
fun hd (Cons f) = #1 (f ()
fun tl (Cons f) = #2 (f ())
```
Task 3 Define the function `fromLoop : (α → α × nat) → α → stream`, which takes a value `v` of type `α` and a function `f` of type `α → α × nat`, successively applies `f` to `v` to get values of type `nat`, and constructs a stream from these natural numbers.

Solution: In ML:

```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 = (α → α × nat) → α → stream`:

```fpc
fromLoop = fix x : t.λf:α→α×nat.λv:α.fold[α.unit→nat×α](λ_.unit.(π2(f v),x f (π1(f v))))
```

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:

```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:

```fpc
natstrk = λk:nat.fromLoop (λv:nat.(s(v),v)) k
natstr = natstrk z
```

Task 5 Define the function, `map : (nat → nat) → stream → stream`, which takes a function `f` and stream `s` and applies `f` to every element in the stream `s`.

Solution: In ML:

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

Translating, writing `t = (nat → nat) → stream → stream`:

```fpc
map = fix x : t.λf:nat→nat.λs:stream.fold[α.unit→nat×α](λ_.unit.(f (hd s),x f (π1(f s))))
```

Task 6 Define the function `streamfix : (stream → stream) → 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`:

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

Instead, we must manually “unroll” the stream once:

```ml
fun streamfix f = Cons (fn () => (hd (f (streamfix f)),tl (f (streamfix f))))
```
Translating, writing \( t = (\text{stream} \rightarrow \text{stream}) \rightarrow \text{stream} \):

\[
\text{streamfix} = \text{fix} : \text{t}.\text{stream} \rightarrow \text{stream}.
\]

\[
\text{fold}(\alpha.\text{unit} \rightarrow \text{nat} \times \alpha)(\lambda_\text{unit}.\langle \text{hd} (f (x \_ f)), \text{tl} (f (x \_ f)) \rangle)
\]

**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:

\[\text{val natstr = streamfix (fn s => Cons (fn _ => (0, map (fn n => n+1) s))});\]

Behavior:

\[\text{val it = 0: int}\]

\[\text{val it = 3: int}\]

Translating:

\[\text{suc} = \lambda n: \text{nat}. s(n)\]

\[\text{natstr2} = \text{streamfix} (\lambda s: \text{stream}. \text{fold}(\alpha.\text{unit} \rightarrow \text{nat} \times \alpha)(\lambda_\text{unit}.\langle \text{z, map suc} s \rangle))\]

**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 \(\text{hd}\) or \(\text{tl}\) is called on it.

\[\text{val foo = streamfix (fn s => s)}\]

\[\text{val foo = Cons fn: stream}\]

\[\text{hd foo}\]

\[\text{(* diverges *)}\]

\[\text{tl foo}\]

\[\text{(* diverges *)}\]

3 Monadization

**Task 9** Define \( \tau \) inductively for each expression \( e \) in \( \text{L1} \).

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

\[\text{Input} = \text{input} \quad \text{output}(e) = \text{bnd}(\tau; y.\text{output}(y))\]

\[\tau = \text{ret}(x) \quad \overline{\tau} = \text{ret}(\pi) \quad \overline{\lambda x: \tau. e} = \text{ret}(\lambda x: \tau. \text{cmd}(\tau))\]

\[\overline{\text{ifz}(e_1; e_2; x.e_3)} = \text{bnd}(\overline{e_1}; x_1. \text{force}(\text{ifz}(x_1; \text{cmd}(\overline{e_2})); x. \text{cmd}(\overline{e_3}))))\]

\[\overline{e_1 e_2} = \text{bnd}(\overline{e_1}; x_1. \text{bnd}(\overline{e_2}; x_2. \text{force}(x_1 x_2)))\]

We assume that products are eager:

\[\overline{(e_1, e_2)} = \text{bnd}(\overline{e_1}; x_1. \text{bnd}(\overline{e_2}; x_2. \text{ret}((x_1, x_2))))\]
\[ \pi_1 e = \text{bnd}(\tau; x.\text{ret}(\pi_1 x)) \quad \pi_2 e = \text{bnd}(\tau; x.\text{ret}(\pi_2 x)) \]

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:

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

My original solution diverges:

\[
\text{fix } x : \tau. e = \text{force}(\text{fix } f : \tau. \text{cmd}(\text{cmd}(\text{force}(f); x.\tau)))
\]